Chapter 11

Optimization I: Greedy Algorithms

In this chapter and the next, we consider algorithms for optimization problems. We have already seen an example of an optimization problem — the maximum subsequence sum problem from Chapter 1. We can characterize optimization problems as admitting a set of candidate solutions. In the maximum subsequence sum problem, the candidate solutions are the contiguous subsequences in the input array. An objective function then typically maps these candidate solutions to numeric values. The objective function for the maximum subsequence sum problem maps each contiguous subsequence to its sum. The goal is to find a candidate solution that either maximizes or minimizes, depending on the problem, the objective function. Thus, the goal of the maximum subsequence problem is to find a candidate solution that maximizes the objective function.

In this chapter, we will examine optimization problems which admit greedy solutions. A greedy algorithm builds a specific candidate solution incrementally. The aspect of a greedy algorithm that makes it “greedy” is how it chooses from among the different ways of incrementing the current partial solution. In general, the different choices are ordered according to some criterion, and the best choice according to this criterion is taken. Thus, the algorithm builds the solution by always taking the step that appears to be most promising at that moment. Though there are many problems for which greedy strategies do not produce optimal solutions, when they do, they tend to be quite efficient. In the next chapter, we will examine a more general technique for solving optimization problems when greedy strategies fail.
11.1 Job Scheduling

Consider the job scheduling problem discussed in Chapter 8. Recall that we are given \( n \) jobs, each requiring one unit of execution time and having its own deadline. Suppose that, in addition, each job has a positive integer value. We wish to schedule the jobs on a single server so as to maximize the total value of those jobs which meet their deadlines. Because jobs which do not meet their deadlines do not contribute any value, we will assume that no jobs are scheduled after their deadlines — if a job can’t meet its deadline, we simply don’t schedule it. At this point, we are not assuming any particular scheduling strategy, such as the one given in Chapter 8; instead, we are trying to find an optimal strategy.

In deriving a greedy algorithm in a top-down fashion, the first step is to generalize the problem so that a partial solution is given as input. We assume as a precondition that this partial solution can be extended to an optimal solution. Our task is then to extend it in some way so that the resulting partial solution can be extended to an optimal solution. If we characterize the size of such an instance as the difference between the size of a complete solution and the given partial solution, we will have reduced a large instance to a smaller instance.

The input to the generalized scheduling problem is a set \( X = \{x_1, \ldots, x_m\} \) of jobs and a partial schedule \( \text{sched} \) of these jobs. To be more precise, let \( \text{sched}[1..n] \) be an array of natural numbers such that if \( \text{sched}[t] = 0 \), then no job has been scheduled in the time slot ending at time \( t \); otherwise, if \( \text{sched}[t] = i \), then job \( x_i \) is scheduled in this time slot. If all the jobs in \( X \) either have been scheduled or cannot be scheduled, we are finished — the precondition that this schedule can be extended to an optimal schedule implies that it must be an optimal schedule. Otherwise, our task is to schedule some job \( x_i \) so that the resulting partial schedule can be extended to a schedule of maximum value. If we take the size of a partial schedule to be the number of unscheduled jobs in \( X \), we will have reduced a large instance to a smaller instance.

We must now decide upon the criterion to use to extend a partial schedule. Of the remaining jobs that can meet their deadlines, it would make sense to schedule the one with the highest value. Furthermore, in order to impact the fewest deadlines of other jobs, it would make sense to schedule it as late as possible. In what follows, we will show that this selection criterion always results in an optimal schedule.

In order to simplify reasoning about this strategy, let us observe that because we will not be changing any scheduling decisions that have already
been made, the values of the jobs scheduled so far have no effect on future decisions — their values are simply added to the total value of the schedule. As a result, all we really need to know about the schedule constructed so far is what time slots are still available. Furthermore, maximizing the values of jobs scheduled in the remaining slots will maximize the total value, because the values of all scheduled jobs are simply added together.

We can therefore focus our attention on the following version of the problem. The input consists of a set $X$ of (unscheduled) jobs and an array $\text{avail}[1..n]$ of boolean values. A valid schedule either assigns a job $x_i$ into a time slot $t$ such that $t$ is no more than the deadline of $x_i$ and $\text{avail}[t] = \text{true}$, or it does not schedule $x_i$. The goal is to maximize the total value of scheduled jobs. The following theorem shows that an optimal schedule can be constructed by selecting the job with maximum value and scheduling it at the latest possible time, assuming it can be scheduled.

**Theorem 11.1** Let $X = \{x_1, \ldots, x_m\}$ be a set of jobs, and let $\text{avail}[1..n]$ be an array of boolean values indicating the time slots at which jobs may be scheduled. Let $x_k$ be a job having maximum value, and suppose there is some $t$ no greater than the deadline of $x_k$ such that $\text{avail}[t] = \text{true}$. Let $t_0$ be the maximum such $t$. Then there is an optimal schedule in which $x_k$ is scheduled at the time slot ending at time $t_0$.

**Proof:** Let $\text{sched}[1..n]$ be an optimal schedule and suppose $\text{sched}[t_0] \neq k$. We consider two cases.

**Case 1:** $\text{sched}[t_1] = k$. Because $t_0$ is the latest available time slot for $x_k$, $t_1 < t_0$. Therefore, by swapping the values of $\text{sched}[t_1]$ and $\text{sched}[t_0]$, we violate no deadlines and do not change the value of the schedule. The resulting schedule must therefore be optimal.

**Case 2:** $x_k$ is not scheduled in $\text{sched}$. Let $j = \text{sched}[t_0]$. We first observe that $j \neq 0$, because in this case we could obtain a schedule with higher value by scheduling $x_k$ in $\text{sched}[t_0]$. Because $x_k$ is a job having maximum value, the value of $x_k$ is at least the value of $x_j$. Therefore, by scheduling $x_k$ at $\text{sched}[t_0]$ instead of $x_j$, we retain an optimal schedule. □

Theorem 11.1 tells us that our greedy strategy results in an optimal schedule. To implement this strategy, we need to consider the jobs in nonincreasing order of their values, and schedule each schedulable job at the latest time possible. Therefore, we should first sort the jobs in nonincreasing order
of their values. Using heap sort or merge sort, this can be done in $\Theta(m\lg m)$ time. **Schedule**, shown in Figure 8.2, then implements the greedy strategy. Because **Schedule** can be implemented to run in $O(n + m\lg n)$ time, if $m \in \Theta(n)$, the entire algorithm runs in $\Theta(n\lg n)$ time.

### 11.2 Minimum-Cost Spanning Trees

Suppose we wish to construct a communications network connecting a given set of nodes. Given the distances separating each pair of nodes, we wish to find the network topology that connects all of the nodes using as little cable as possible.

We can state the above problem as a graph problem. In Exercise 9.6, we defined a tree to be connected, acyclic, undirected graph. (Note that a tree is different from a rooted tree as defined on page 153, though we can form a rooted tree from a tree by selecting any vertex as the root.) Given a connected undirected graph $G = (V, E)$, a spanning tree is a tree $(V, T)$ such that $T \subseteq E$; i.e., a spanning tree is a tree consisting of all of the vertices of $G$ and a subset of the edges. Let $\text{cost} : E \rightarrow \mathbb{N}$ give a cost for each edge. We wish to find a minimum-cost spanning tree (MST) for $G$ — i.e., a spanning tree whose edges have minimum total cost.

In order to develop a greedy algorithm, we first generalize the problem so that a portion of an MST is given as input. This partial MST will be a subset $E' \subseteq E$ such that $(V, E')$ is acyclic, but not necessarily connected. In order to keep the cost as small as possible, we will use as our selection criterion the cost of the edge; i.e., we will always select a least-cost edge that does not introduce a cycle.

We need to show that the above strategy will result in an MST. In order to state the theorem that guarantees this fact, we need one definition. Let $G = (V, E)$ be an undirected graph. A connected component of $G$ is any connected subset $C \subseteq V$ such that no vertex in $C$ is adjacent to any vertex in $V \setminus C$. Thus, the connected component containing a vertex $v$ is the set of all vertices reachable from $v$ using zero or more edges. We can now state the following theorem, which validates our selection strategy.

**Theorem 11.2** Let $G = (V, E)$ be a connected undirected graph with cost function $\text{cost} : E \rightarrow \mathbb{N}$, and let $E' \subseteq E$ be such that for some MST $(V, T)$ of $G$, $E' \subseteq T$. Suppose that $(V, E')$ is not connected, and let $C$ be a connected component of $(V, E')$. If $\{u, v\} \in E \setminus E'$ is a minimum-cost edge such that $u \in C$ and $v \not\in C$, then there is an MST of $G$ containing all the edges in $E' \cup \{\{u, v\}\}$. 

Before we prove Theorem 11.2, we note that it is stronger than we require. We state it in this way in order to justify a second greedy algorithm, which we will discuss a bit later. Note, however, that a minimum-cost edge that does not introduce a cycle certainly qualifies as the edge \{u, v\} in the statement of the theorem.

**Proof of Theorem 11.2:** If \(\{u, v\} \in T\), then there is nothing to show. Suppose \(\{u, v\} \notin T\). We will show how to construct a set \(T'\) such that \(E' \cup \{\{u, v\}\} \subseteq T'\) and \((V, T')\) is an MST.

Because \((V, T)\) is a tree, there is a path from \(u\) to \(v\) in \(T\). However, this path cannot contain the edge \(\{u, v\}\), because \(\{u, v\} \notin T\). This path must therefore consist of:

- a (possibly empty) path in \(C\) from \(u\) to some vertex \(w\);
- an edge \(\{w, x\}\), where \(x \notin C\); and
- a (possibly empty) path from \(x\) to \(v\).

Note that even though either path above might be empty, they cannot both be empty, or we would have \(\{w, x\} = \{u, v\}\). By the choice of \(\{u, v\}\), \(\text{cost}(\{w, x\}) \geq \text{cost}(\{u, v\})\). Let \(T' = (T \cup \{\{u, v\}\}) \setminus \{\{w, x\}\}\). Then the total cost of \(T'\) is no more than the total cost of \(T\). From Exercise 9.6 a, \(T\), and hence \(T'\), has exactly \(|V| - 1\) edges. Furthermore, \((V, T')\) is connected, as any path in \(T\) that contains \(\{w, x\}\) can be modified to use the edge \(\{u, v\}\) and the paths from \(u\) to \(w\) and \(x\) to \(v\) in \(T\). From Exercise 9.6 b, \((V, T')\) is a tree. Because its cost is no more than the cost of the MST \((V, T)\), \((V, T')\) is an MST containing all of the edges in \(E' \cup \{\{u, v\}\}\). \(\square\)

In order to implement this algorithm, we need an efficient way to determine whether two vertices belong to the same connected component of \((V, E')\), where \(E'\) is the set of edges we have collected so far. The connected components form disjoint subsets of \(V\). We can therefore maintain these connected components using a DISJOINTSETS structure. In order to determine whether \(u\) and \(v\) belong to the same component, we see if \(\text{FIND}(u) = \text{FIND}(v)\). If not, we include \(\{u, v\}\) and merge the two components.

The algorithm, known as Kruskal’s algorithm, is shown in Figure 11.1. Note that in using an INVERTEDPRIORITYQUEUE, we process the edges in nondecreasing order of cost. We could have achieved the same effect by sorting the edges by cost, but this presentation is somewhat simpler, and in fact amounts to sorting the edges with heap sort.
Figure 11.1 Kruskal’s algorithm for finding an MST

**Precondition:** $G$ refers to a Graph which is undirected and connected, and whose edges contain their costs as data.

**Postcondition:** Returns a ConsList of Edges representing an MST of $G$. Each edge will occur once in the list.

**KRUSKAL($G$)**

$n \leftarrow G.Size();$  
$comp \leftarrow \text{new DisjointSets}(n)$  
$q \leftarrow \text{new InvertedPriorityQueue}();$  
$T \leftarrow \text{new ConsList}()$

for $i \leftarrow 0$ to $n - 1$

$L \leftarrow G.AllFrom(i)$

while not $L.ISEmpty()$

  $e \leftarrow L.Head();$  
  $q.Put(e, e.Data());$  
  $L \leftarrow L.Tail()$

  // **Invariant:** $T$ is a subset of the edges of an MST of $G$, the sets in $\text{comp}$ are the connected components of $T$, and $q$ contains a subset of the edges of $G$ ordered by cost, including at least all \{$u,v$\} such that $u$ and $v$ are in different sets in $\text{comp}$.

while not $q.ISEmpty()$

  $e \leftarrow q.RemoveMin()$

  $c_1 \leftarrow \text{comp.Find}(e.Source());$  
  $c_2 \leftarrow \text{comp.Find}(e.Dest())$

  if $c_1 \neq c_2$

    $T \leftarrow \text{new ConsList}(e, T);$  
    $\text{comp.Merge}(c_1, c_2)$

  return $T$

For the purpose of analyzing **KRUSKAL**, let $n$ be the number of vertices in $G$, and let $a$ be the number of edges. Let us first assume that $G$ is implemented using **ListGraph**. In the initialization code preceding the first loop, $\Theta(n)$ time is required to construct a new **DisjointSets** structure, whereas the other operations each run in $\Theta(1)$ time. The **for** loop with the nested **while** simply traverses $G$ in a manner similar to the loops found in **TopSort** (Figure 9.6). If we ignore for the moment the cost of the **Put** operations on $q$, we see that this nested structure runs in $\Theta(n+a)$ time. Each edge is inserted into $q$; hence, because **InvertedPriorityQueue**.Put runs in $\Theta(lg \ i)$ time when there are $i$ elements in the queue, the total time for all insertions is in $\Theta(a \ lg \ a)$. Because $G$ is connected, $n - 1 \leq a \leq n(n - 1)/2$. Furthermore, $\lg(n(n-1)/2) < 2 \lg n$, so that $\Theta(n+a) + \Theta(a \ lg \ a) = \Theta(a \ lg \ n)$. The last **while** loop is easily seen to run in $\Theta(a \ lg \ n)$ time as well.
If $G$ is implemented using MatrixGraph, the AllFrom operation requires $\Theta(n)$ time, so that the for loop requires $\Theta(n^2)$. The total running time is therefore in $\Theta(n^2 + a \lg n)$, which is worse than $\Theta(a \lg n)$ for sufficiently sparse graphs (i.e., when $a \in o(n^2/\lg n)$). Kruskal’s algorithm therefore tends to be better-suited for the ListGraph implementation, particularly when the graph is sparse.

As we suggested earlier, Kruskal’s algorithm isn’t the only greedy algorithm for finding MSTs. We arrive at a different algorithm if we generalize the original problem in a slightly different way. Rather than allowing our input to consist of any set of edges that can be extended to an MST, we instead require that this set of edges form a spanning tree on some subset of the vertices. Thus, when we add an edge, it must extend this spanning tree to another vertex; i.e., it must connect a vertex in the spanning tree to one that is not in the spanning tree. Our selection criterion will be to select such an edge having minimum cost. Theorem 11.2 tells us that such a strategy results in an MST.

The data structures needed to implement this algorithm, which is known as Prim’s algorithm, are simpler than those needed to implement Kruskal’s algorithm. We need to partition the vertices into two disjoint sets — the set of vertices in the spanning tree and those not in the spanning tree. A boolean array $\text{inTree}[0..n - 1]$ will suffice for this purpose. For each vertex $k$ not in the spanning tree, we need an efficient way to find a least-cost edge $\{i, k\}$ such that $i$ is in the spanning tree. For this purpose, we use two arrays:

- an array $\text{bestCost}[1..n - 1]$ such that if $k$ is not in the spanning tree, then $\text{bestCost}[k]$ is the minimum cost of any edge to $k$ from a vertex in the spanning tree, or $\infty$ if there is no such edge; and
- an array $\text{best}[1..n - 1]$ such that if $k$ is not in the spanning tree and $\text{bestCost}[k] \neq \infty$, then $\{\text{best}[k], k\}$ is a least-cost edge from the spanning tree to $k$.

The spanning tree will initially contain only the vertex 0; hence, it is unnecessary to include the index 0 for the arrays $\text{best}$ and $\text{bestCost}$. We can then initialize each $\text{best}[k]$ to 0 and each $\text{bestCost}[k]$ to the cost of edge $\{0, k\}$, or to $\infty$ if there is no such edge. In order to find an edge to add to the spanning tree we can find the minimum $\text{bestCost}[k]$ such that $k$ is not in the spanning tree. If we denote this index by $\text{next}$, then the edge $\{\text{best}[\text{next}], \text{next}\}$ is the next edge to be added, thus connecting $\text{next}$ to the spanning tree. For each $k$ that is still not in the spanning tree, we must
then update \( \text{bestCost}[k] \) by comparing it to the cost of \( \{\text{next}, k\} \), and update \( \text{best}[k] \) accordingly. The algorithm is shown in Figure 11.2.

It is easily seen that if \( G \) is a MatrixGraph, the running time is in \( \Theta(n^2) \). This is an improvement over Kruskal’s algorithm when a MatrixGraph is used. If a ListGraph is used, however, the running time is still in \( \Omega(n^2) \), and can be as bad as \( \Theta(n^3) \) for dense graphs. Thus, Kruskal’s algorithm is preferred when a ListGraph is used, but Prim’s algorithm is preferred when a MatrixGraph is used. If we have the freedom of choosing the Graph implementation, we should choose a ListGraph and Kruskal’s algorithm for sparse graphs, but a MatrixGraph and Prim’s algorithm for dense graphs.

### 11.3 Single-Source Shortest Paths

Suppose we wish to drive from Los Angeles to Boston. A cursory glance at a road atlas tells us there are many different routes we can take. We decide that we wish to take a route that optimizes a particular objective function — perhaps total length or total expected time. We can model such a problem as an instance of a shortest path problem. We model the various road segments as edges of a directed graph \( G \) whose vertices are the road intersections. We represent Los Angeles with a start vertex \( u \) and Boston with an end vertex \( v \). A cost (representing either length or expected time) is associated with each edge. We wish to find a least-cost path from \( u \) to \( v \) in \( G \).

We will use the term *length* to refer to the cost of an edge, regardless of what kind of cost is being represented. A least-cost path is then a shortest path. We will assume that edge lengths are positive integers. Note that if we have found a shortest path from \( u \) to \( v \), and if vertex \( w \) occurs on this path, then the subpath from \( u \) to \( w \) is also a shortest path from \( u \) to \( w \). As a result, when we find a shortest path from \( u \) to \( v \), we typically find many other shortest paths from \( u \) as well. For this reason, it simplifies the discussion to generalize the problem to that of finding, for every vertex \( w \), a shortest path from \( u \) to \( w \).

We first observe that for each vertex \( w \neq u \), if we consider only a single shortest path from \( u \) to \( w \), then there is a unique predecessor \( x \) of \( w \) on this path. Furthermore, we can select the shortest paths in such a way that \( x \) precedes \( w \) on any shortest path on which \( w \) occurs. Thus, for each vertex in \( G \), there is a unique sequence of these predecessors leading back to \( u \). This predecessor relationship therefore gives the parent relationship.
Precondition: $G$ refers to a Graph which is undirected and connected, and whose edges contain their costs as data.

Postcondition: Returns a ConsList of Edges representing an MST of $G$. Each edge will occur once in the list.

PRIM($G$)

$n \leftarrow G.\text{Size}();\ \text{inTree} \leftarrow \text{new Array}[0..n-1];\ \text{inTree}[0] \leftarrow \text{true}$

$\text{best} \leftarrow \text{new Array}[1..n-1];\ \text{bestCost} \leftarrow \text{new Array}[1..n-1]$

$T \leftarrow \text{new ConsList}()$

for $k \leftarrow 1\ \text{to}\ n-1$

$\text{inTree}[k] \leftarrow \text{false};\ \text{best}[k] \leftarrow 0;\ \text{bestCost}[k] \leftarrow G.\text{Get}(0,k)$

if $\text{bestCost}[k] = \text{nil}$

$\text{bestCost}[k] \leftarrow \infty$

// Invariant: $T$ contains count edges forming a spanning tree for the
// vertices $k$ such that $\text{inTree}[k]$ is true, and there is an MST of $G$
// containing all of the edges in $T$. For each $k$ such that $\text{inTree}[k]$ is false,
// $\text{bestCost}[k]$ is the minimum cost of any edge $\{i, k\}$ such that $\text{inTree}[i]$
// is true, or $\infty$ if there is no such edge. For each $k$ such that $\text{inTree}[k]$
// is false and $\text{bestCost}[k] \neq \infty$, $\{\text{best}[k], k\}$ is a least-cost edge
// leading to $k$ from any vertex $i$ such that $\text{inTree}[i]$ is true.

for $\text{count} \leftarrow 0\ \text{to}\ n-2$

$m \leftarrow \infty$

for $k \leftarrow 1\ \text{to}\ n-1$

if not $\text{inTree}[k] \text{ and } \text{bestCost}[k] < m$

$\text{next} \leftarrow k;\ m \leftarrow \text{bestCost}[k]$

$e \leftarrow \text{new Edge(}\text{best}[\text{next}],\ \text{next},\ m)\$

$T \leftarrow \text{new ConsList}(e, T);\ \text{inTree}[\text{next}] \leftarrow \text{true}$

for $k \leftarrow 1\ \text{to}\ n-1$

if not $\text{inTree}[k]$

$d \leftarrow G.\text{Get}(\text{next}, k)$

if $d \neq \text{nil} \text{ and } d < \text{bestCost}[k]$

$\text{best}[k] \leftarrow \text{next};\ \text{bestCost}[k] \leftarrow d$

return $T$

---

**Figure 11.2** Prim’s algorithm for finding an MST
of a tree rooted at \( u \). This rooted tree can be used to represent the shortest paths.

Let us now generalize the problem so that a tree \( T \) rooted at \( u \) and containing a subset of the edges and vertices of the graph is provided as additional input. Suppose that this tree is a proper subtree of a shortest path tree; i.e., suppose that for each vertex \( w \) in the tree, the path from \( u \) to \( w \) in the tree is a shortest path from \( u \) to \( w \) in \( G \). We need to add a vertex \( x \) and an edge \((w,x)\), where \( w \) is a vertex in \( T \), so that the path from \( u \) to \( x \) in the resulting tree is a shortest path in \( G \) from \( u \) to \( x \).

For each vertex \( w \) in \( T \), let \( d_w \) give the length of the path from \( u \) to \( w \) in \( T \). For each edge \((x,y)\) in \( G \), let \( \text{len}(x,y) \) give the length of \((x,y)\). Let \((w,x)\) be an edge in \( G \) such that

- \( w \) is in \( T \);
- \( x \) is not in \( T \); and
- \( d_w + \text{len}(w,x) \) is minimized.

Clearly, the path from \( u \) to \( w \) in \( T \), followed by the edge \((w,x)\) is a shortest path from \( u \) to \( x \) in \( G \), as there can be no shorter paths from \( u \) to any vertex not in \( T \).

Building a shortest path tree in this way is very similar to the way Prim’s algorithm builds an MST. While Prim’s algorithm is for undirected graphs, this algorithm — Dijkstra’s algorithm — is for directed graphs. The computation for updating the \text{bestCost} array must use \( d_w + \text{len}(w,x) \), rather than the cost of \((w,x)\) (we assume that \( \text{len}(w,x) \) will be stored in the \text{data} variable for edge \((w,x)\)). Hence, the value \( d_w \) must be computed and stored as vertices are added to the tree. The resulting algorithm runs in \( \Theta(n^2) \) time, the same as for Prim’s algorithm; we leave the details as an exercise.

### 11.4 Huffman Codes

Compression algorithms are often used for data archival or for improving data transmission rates. In this section, we examine one of the key components of data compression. In order to simplify the discussion, we will assume we are dealing with character data, though the techniques apply more generally.

Typical character encodings, such as ASCII or Unicode, are fixed-width encodings — all characters are encoded using the same number of bits. However, in a typical natural-language text, some characters like ‘e’ occur
much more frequently than other characters like ‘X’. It might make sense, then, to use a variable-width encoding, so that the more frequently occurring characters have shorter codes. Furthermore, if the encoding were chosen for a particular text document, we could use shorter codes overall because we would only need to encode those characters that actually appear in the document.

The difficulty with variable-width encodings is choosing the encoding so that it is clear where one character ends and the next begins. For example, if we encode ‘n’ with 11 and ‘o’ with 111, then the encoding 11111 would be ambiguous — it could encode either “no” or “on”. To overcome this difficulty, we arrange the characters as the leaves of a binary tree in which each non-leaf has two nonempty children (see Figure 11.3). The encoding of a character is determined by the path from the root to the leaf containing that character: each left child on the path denotes a 0 in the encoding, and each right child on the path denotes a 1 in the encoding. Thus, in Figure 11.3, ‘M’ is encoded as 100. Because no path from the root to a leaf is a proper prefix of any other path from the root to a leaf, no ambiguity results.

Example 11.3 For example, we can use the tree in Figure 11.3 to encode “Mississippi” as 100011110111101011010. We parse this encoding by traversing the tree according to the paths specified by the encoding. Starting at the root, we go right-left-left, arriving at the leaf ‘M’. Starting at the root again, we go left, arriving at the leaf ‘i’. Continuing in this manner, we
see that the bit-string decodes into “Mississippi”. Note that because there are four distinct characters, a fixed-width encoding would require at least two bits per character, yielding a bit string of length 22. However, the bit string produced by the given encoding has length 21.

The specific problem we wish to address in this section is that of producing a tree that yields a minimum-length encoding for a given text. We will not concern ourselves with the counting of the characters in the text; rather, we will assume that a frequency table has been produced and is provided as our input. This frequency table gives the number of occurrences of each character in the text. To simplify matters, we will assume that none of the characters in the table has a frequency of 0. Furthermore, we will not concern ourselves with producing the encoding from the tree; i.e., our output consists solely of a binary tree storing the information we need in order to extract each character’s code.

We first need to consider how we can determine the length of an encoded string for a particular encoding tree. Note that when we decode a bit string, we traverse exactly one edge in the tree for each bit of the encoding. One way to determine the length of the encoding is therefore to compute the number of times each edge would be traversed during decoding. A given edge \((u, v)\) is traversed once for each occurrence of each character in the subtree rooted at \(v\). For a subtree \(t\), let us therefore define \(\text{weight}(t)\) to be the total number of occurrences of all characters in \(t\). For an encoding tree \(T\), we can then define \(\text{cost}(T)\) to be the sum of the weights of all proper subtrees of \(T\). (Note that \(\text{weight}(T)\) will always be the length of the given text.) \(\text{cost}(T)\) then gives the length of the encoding based on \(T\). For a given frequency table, we define a Huffman tree to be an encoding tree with minimum cost for that table.

Let us now generalize the problem so that the input is a collection of trees, \(t_1, \ldots, t_n\), each of which encodes a portion of the frequency table. We assume that each character in the frequency table occurs in exactly one of the input trees, and that the frequency table has a Huffman tree that contains all of the input trees as subtrees. Note that if all of the trees are single nodes, this input is just the information from the frequency table. If the input consists of more than one tree, we need to merge two of the trees by making them the children of a new root. Furthermore, we need to be able to do this so that the frequency table has a Huffman tree containing all of the resulting trees as subtrees. We claim that merging two trees of minimum weight will produce such a tree.
Theorem 11.4 Let \( T \) be a Huffman tree for a frequency table \( F \), and let \( t_1, \ldots, t_n \) be subtrees of \( T \) such that \( n > 1 \) and each leaf of \( T \) occurs in exactly one of \( t_1, \ldots, t_n \). Suppose \( \text{weight}(t_1) \leq \text{weight}(t_2) \leq \cdots \leq \text{weight}(t_n) \). Let \( t_{n+1} \) be the binary tree formed by making \( t_1 \) and \( t_2 \) the left and right children, respectively, of a new root. Then there is a Huffman tree \( T' \) for \( F \) containing \( t_3, t_4, \ldots, t_{n+1} \) as subtrees.

Proof: If \( t_{n+1} \) is a subtree of \( T \), then we can let \( T' = T \). Furthermore, if \( T \) has a subtree with \( t_1 \) as the right child and \( t_2 \) as the left child, we can simply swap \( t_1 \) with \( t_2 \) and let the resulting tree be \( T' \). Otherwise, \( t_1 \) and \( t_2 \) are not siblings in \( T \). Furthermore, neither can be a subtree of the other because they have no leaves in common. Let node \( x \) be the lowest common ancestor of \( t_1 \) and \( t_2 \) in \( T \); i.e., \( x \) is an ancestor of both \( t_1 \) and \( t_2 \), but neither child of \( x \) is. We consider two cases.

Case 1: The path from \( x \) to \( t_1 \) is no longer than the path from \( x \) to \( t_2 \). Let \( t \) be the sibling of \( t_2 \) in \( T \). Without loss of generality, assume \( t \) is the left child and \( t_2 \) is the right child (otherwise, we can swap them). Clearly, \( t \) can be neither \( t_1 \) nor \( t_2 \). Furthermore, it cannot be a proper subtree of any of \( t_1, \ldots, t_n \), because then \( t_2 \) would also be a proper subtree of the same tree. Finally, \( t \) cannot contain \( t_1 \) as a proper subtree, because then the path from \( x \) to \( t_1 \) would be longer than the path from \( x \) to \( t_2 \). We conclude that \( t \) must contain one or more of \( t_3, \ldots, t_n \). We can therefore swap \( t_1 \) with \( t \), letting the result be \( T' \).

Because \( t \) contains one or more of \( t_3, \ldots, t_n \), \( \text{weight}(t_1) \leq \text{weight}(t) \); hence, \( \text{weight}(t) - \text{weight}(t_1) \geq 0 \). The swap then causes the weights of all nodes except \( x \) on the path from \( x \) to the parent of \( t_1 \) in \( T \) to increase by \( \text{weight}(t) - \text{weight}(t_1) \). Furthermore, it causes the weights of all nodes except \( x \) on the path from \( x \) to the parent of \( t_2 \) in \( T \) to decrease by \( \text{weight}(t) - \text{weight}(t_1) \). No other nodes change weight. Because there are at least as many nodes on the path from \( x \) to \( t_2 \) in \( T \) as on the path from \( x \) to \( t_1 \) in \( T \), the swap cannot increase the cost of the tree. Therefore \( T' \) is a Huffman tree.

Case 2: The path from \( x \) to \( t_1 \) is longer than the path from \( x \) to \( t_2 \). In this case we assume without loss of generality that \( t_1 \) is a left child, and we swap its sibling with \( t_2 \). Because Case 1 doesn’t rely on the fact that \( \text{weight}(t_1) \leq \text{weight}(t_2) \), the same reasoning holds for this case.

We assume the frequency table is provided via two arrays:
Figure 11.4 Algorithm for constructing a Huffman tree

**Precondition:** `chars[1..n]` is an array of `Chars`, `n ≥ 1`, and `freq[1..n]` is an array of positive `Nats`.

**Postcondition:** Returns a `BinaryTreeNode` representing a Huffman tree for text including `freq[i]` occurrences of `char[i]`, for `1 ≤ i ≤ n`.

```
HuffmanTree(chars[1..n], freq[1..n])
    q ← new InvertedPriorityQueue()
    for i ← 1 to n
        t ← new BinaryTreeNode(); t.SetRoot(chars[i])
        q.Put(t, freq[i])
    // Invariant: q contains BinaryTreeNodees which are subtrees of
    // some Huffman tree for the given frequency table, and whose priorities
    // are their weights as defined by the frequencies of the characters
    // contained in their leaves. Each character in chars[1..n] occurs in
    // exactly one leaf of one tree.
    while q.Size() > 1
        w1 ← q.MinPriority(); t1 ← q.RemoveMin()
        w2 ← q.MinPriority(); t2 ← q.RemoveMin()
        t ← new BinaryTreeNode(); t.SetLeft(t1); t.SetRight(t2)
        q.Put(t, w1 + w2)
    return q.RemoveMin()
```

- `chars[1..n]`, which contains the characters in the table; and
- `freq[1..n]`, which contains positive integers giving the frequencies of the corresponding characters.

The algorithm should then return a `BinaryTreeNode` representing a Huffman tree for the frequency table. The data in the leaves are characters, and all other data items in the tree are `nil`. The algorithm is shown in Figure 11.4. We maintain the trees in an `InvertedPriorityQueue` using the weights of the trees as priorities.

Because each iteration of the `for` loop adds an element to an initially empty priority queue, iteration `i` runs in $\Theta(\lg i)$ time. The `for` loop therefore runs in $\Theta(n \lg n)$ time. After the `for` loop completes, `q` contains `n` elements. Each iteration of the `while` loop removes two elements from `q` and adds one
element. The loop therefore iterates \( n - 1 \) times. Each iteration runs in \( O(\lg n) \) time. The running time of the \textbf{while} loop is therefore in \( O(n \lg n) \), so that the algorithm runs in \( \Theta(n \lg n) \) time.

### 11.5 Summary

Greedy algorithms provide an efficient mechanism for solving certain optimization problems. The major steps involved in the construction of a greedy algorithm are:

- Generalize the problem so that a partial solution is given as input.
- Decide upon a selection criterion for incrementally extending partial solutions.
- Prove that if a given partial solution can be extended to an optimal solution, then after extending this partial solution using the chosen selection criterion, the resulting partial solution can also be extended to an optimal solution.
- Implement the transformation suggested by the incremental extension using a loop.

Priority queues are often useful in facilitating quick access to the best extension, as determined by the selection criterion. In many cases, the extension involves joining pieces of a partial solution in a way that can be modeled effectively using a \texttt{DisjointSets} structure.

Proving that the incremental extension can be extended to an optimal solution is essential, because it is not true for all selection criteria. In fact, there are optimization problems for which there is no greedy solution. In the next chapter, we will examine a more general, though typically more expensive, technique for solving optimization problems.

### 11.6 Exercises

**Exercise 11.1** Prove that \texttt{Kruskal}, shown in Figure 11.1, meets its specification.

**Exercise 11.2** Prove that \texttt{Prim}, shown in Figure 11.2, meets its specification.
Exercise 11.3 Instead of using the arrays `best` and `bestCost`, Prim’s algorithm could use a priority queue to store all of the edges from vertices in the spanning tree. As vertices are added to the spanning tree, all edges from these vertices would be added to the priority queue. As edges are removed from the priority queue, they would need to be checked to see if they connect a vertex in the spanning tree with one that is not in the spanning tree. Implement this algorithm and analyze its running time assuming the graph is implemented as a `ListGraph`.

Exercise 11.4 Implement the single-source shortest path algorithm as outlined in Section 11.3. Your algorithm should take a `Graph G` and a natural number `u < G.Size()` and return an array `pred[0..G.Size() − 1]` such that `pred[i]` gives the parent of `i` in the shortest paths tree; `pred[u]` should be −1.

Exercise 11.5 Modify your algorithm from Exercise 11.4 to use a priority queue as suggested in Exercise 11.3. Analyze its running time assuming the graph is implemented as a `ListGraph`.

Exercise 11.6 Suppose we wish to solve the single-source shortest path problem for a graph with unweighted edges; i.e., each edge is understood to have a length of 1. Prove that the algorithm for Exercise 11.5 can be modified by replacing the priority queue with a queue (see Exercise 4.11, page 143) to yield an algorithm for the unweighted single-source shortest path problem. Analyze the running time of the resulting algorithm, assuming the graph is implemented as a `ListGraph`. (This algorithm is known as breadth-first search.)

Exercise 11.7 Construct a Huffman tree for the string, “banana split”, and give its resulting encoding in binary. Don’t forget the blank character.

Exercise 11.8 Prove that `HUFFMAN_TREE`, shown in Figure 11.4, meets its specification.

Exercise 11.9 Suppose we have a set of jobs, each having a positive integer execution time. We must schedule all of the jobs on a single server so that at most one job occupies the server at any given time and each job occupies the server for a length of time equal to its execution time. Our goal is to minimize the sum of the finish times of all of the jobs. Design a greedy algorithm to accomplish this and prove that it is optimal. Your algorithm should run in \(O(n \lg n)\) time, where \(n\) is the number of jobs.
CHAPTER 11. OPTIMIZATION I: GREEDY ALGORITHMS

Exercise 11.10 Extend the above exercise to \( k \) servers, so that each job is scheduled on one of the servers.

Exercise 11.11 Suppose we are given a set of events, each having a start time and a finish time. Each event requires a single room. We wish to assign events to rooms using as few rooms as possible so that no two events in the same room overlap (they may, however, be scheduled “back-to-back” with no break in between). Give a greedy algorithm to accomplish this and prove that it is optimal. Your algorithm should run in \( O(n \log n) \) time.

Exercise 11.12 Repeat the above exercise with the constraint that only one room is available. The goal is to schedule as many events as possible.

Exercise 11.13 We wish to plan a trip across country in a car that can go \( d \) miles on a full tank of gasoline. We have identified all of the gas stations along the proposed route. We wish to plan the trip so as to make as few stops for gasoline as possible. Design a greedy algorithm that gives an optimal set of stops when given \( d \) and an array \( \text{dist}[1..n] \) such that \( \text{dist}[i] \) gives the distance from the starting point to the \( i \)th gas station. Your algorithm should operate in \( O(n) \) time.

* Exercise 11.14 The fractional knapsack problem is as follows. We are given a set of \( n \) items, each having a positive weight \( w_i \in \mathbb{N} \) and a positive value \( v_i \in \mathbb{N} \). We are also given a weight bound \( W \in \mathbb{N} \). We wish to carry some of these items in a knapsack without exceeding the weight bound. Our goal is to maximize the total value of the items we carry. Furthermore, the items are such that we can take a fraction of the item if we wish. Thus, we wish to maximize

\[
\sum_{i=1}^{n} a_i v_i
\]

for rational \( a_1, \ldots, a_n \) such that for \( 1 \leq i \leq n \), \( 0 \leq a_i \leq 1 \), and subject to the constraint that

\[
\sum_{i=1}^{n} a_i w_i \leq W.
\]

a. Give a greedy algorithm to find an optimal packing, and prove that your algorithm is correct. Your algorithm should run in \( O(n \log n) \) time.
b. Show using a specific example that this greedy algorithm does not always give an optimal solution if we require that each $a_i$ be either 0 or 1.

c. Using techniques from Chapter 10, improve the running time of your algorithm to $O(n)$.

11.7 Chapter Notes

Greedy algorithms were first identified in 1971 by Edmonds [34], though they actually existed long before then. The theory that underlies greedy algorithms — matroid theory — was developed by Whitney [112] in the 1930s. See, e.g., Lawler [84] or Papadimitriou and Steiglitz [90] for more information on greedy algorithms and matroid theory.

The first MST algorithm was given by Borůvka [16] in 1926. What is now known as Prim’s algorithm was first discovered by Jarník [67], and over 25 years later rediscovered independently by Prim [92] and Dijkstra [27]; the latter paper also includes the single-source shortest paths algorithm outlined in Section 11.3. Kruskal’s algorithm was given by Kruskal [83]. Other MST algorithms have been given by Yao [116], Cheriton and Tarjan [21], and Tarjan [104]. Other improvements for single-source shortest paths have been given by Johnson [71, 72], Tarjan [104], and Fredman and Tarjan [44].

Huffman coding was developed by Huffman [65]. See Lelewer and Hirchberg [85] for a survey of compression algorithms. On the website for this textbook is a tool for constructing and displaying a Huffman tree for a given text.