Lecture V
THE GREEDY APPROACH

An algorithmic approach is called "greedy" when it makes decisions for each step based on what seems best at the current step. Moreover, once a decision is made, it is never revoked. It may seem that this approach is rather limited. Nevertheless, many important problems have special features that allow efficient solution using this approach. An essential point of greedy solutions is that we never have to revise our greedy decisions, and this leads to fast algorithms provided we can make the greedy decision quickly.

The greedy method is supposed to exemplify the idea of "local search". But closer examination of greedy algorithms will reveal some global information being used. Such global information is usually minimal. Typically it amounts to some global sorting step. Indeed, the preferred data structure for delivering this global information is the priority queue.

In this chapter, we consider two problems that use the greedy approach: Huffman tree construction and minimum spanning trees. An abstract setting for the minimum spanning tree problem is based on **matroid theory** and the associated **maximum independent set problem**. We introduce this framework to capture the essence of many problems with greedy solutions.

§1. Linear Bin Packing

In this section, we give a very simple example of greedy algorithms, called linear bin packing. However, it is related to a major topic in algorithms, namely, bin packing problems. The prototype bin packing problem involves putting a set of items into the minimum number of bins. Each item is characterized by its weight, and the bins are identical, with a limited capacity. More precisely, we are given a multiset set $W = \{w_1, \ldots, w_n\}$ of positive weights, and a bin capacity $M > 0$. We want to partition W into a minimum number of subsets ("bins") such that the total weight in each bin is at most M. We may assume that each $w_i \leq M$. E.g., if $W = \{1, 1, 1, 3, 2, 2, 1, 3, 1\}$ and $M = 5$ then one solution is

$$\{3, 2\}, \{2, 3\}, \{1, 1, 1, 1, 1\}.$$

This solution uses 3 bins, and it is clearly optimal, as each bin is filled to capacity. In general, bin packing is considered a hard problem because all known algorithm for optimal bin packing is exponential time. But we can turn hard problems into feasible ones by imposing suitable restrictions. We now illustrate this with a linearized version of bin packing.

Suppose we have a joy ride in an amusement park where riders arrive in a queue. We want to assign riders into cars, where the cars are empty when they arrive. Each car has a weight limit $M > 0$. The number of riders in a car in immaterial, as long as their total weight is $\leq M$ pounds. For instance, if $M = 400$ and the weights (in pounds) of the riders in the queue are $(40, 190, 80, 210, 100, 80, 50, 170)$ then we can put the riders in cars in the following groups:

$$S_1 : (40, 190, 80), (210, 100, 80), (50, 170).$$

Solution S_1 uses three cars (the first car has the first 3 riders, the next car has the next 3, and the last care has 2 riders). It is the solution given by the standard "greedy algorithm". Here are two other non-greedy solutions:

$$S_2 : (40, 190), (80, 210), (100, 80, 50, 170).$$

$$S_3 : (40, 190)(80, 210), (100, 80, 50, 170).$$

Greedy Algorithm. The algorithm has a simple iterative solution. Let C be a container (or car) that is being filled, and let W be the cumulative weight of the elements being added to C. Initially, $W \leftarrow 0$ and $C = \emptyset$. If the next weight would make C overfull, we output C. So here is the code:

```
▷ Initialization
    C ← ∅, W ← 0.
▷ Loop
    for i = 1 to n + 1
        if (i = n + 1 or W + w_i > M)
            Output C, W ← 0, C ← ∅.
        else
            W ← W + w_i, Add w_i to C.
```

At the conclusion of this for-loop, we would have output a sequence car loads representing the greedy solution.

Correctness. It is not obvious why this algorithm produces an optimal bin packing. Here is a proof by induction. Suppose the greedy algorithm outputs k cars with the weights

$$(w_1, \ldots, w_{n_1}), (w_{n_1+1}, \ldots, w_{n_2}), \ldots, (w_{n_{k-1}+1}, \ldots, w_{n_k})$$

where $n_k = n$. This defines a sequence of indices,

$$1 \leq n_1 < n_2 < \cdots < n_k = n.$$

Consider any optimal solution with ℓ cars with the weights

$$(w_1, \ldots, w_{m_1}), (w_{m_1+1}, \ldots, w_{m_2}), \ldots, (w_{m_{\ell-1}+1}, \ldots, w_{m_\ell})$$

where

$$1 \leq m_1 < m_2 < \cdots < m_\ell = n.$$

Since this is optimal, we have

$$\ell \leq k.$$

We claim that for $i = 1, \ldots, \ell$,

$$m_i \leq n_i. \tag{1}$$

It is easy to see that this is true for $i = 1$. For $i > 1$, assume $m_{i-1} \leq n_{i-1}$ by induction hypothesis. By way of contradiction, suppose that $m_i > n_i$. Then

$$m_{i-1} \leq n_{i-1} \leq n_i - 1 \leq m_i - 2.$$

This means that the i-th car on the optimal solution has weights

$$w_{m_{i-1}+1} + \cdots + w_{m_i} > w_{n_{i-1}+1} + \cdots + w_{n_i} + w_{n_i+1}.$$

But by definition of the greedy algorithm, the sum $w_{n_{i-1}+1} + \cdots + w_{n_i} + w_{n_i+1}$ must exceed M (otherwise the greedy algorithm would have added w_{n_i+1} to the ith car). This is a contradiction. This concludes our proof of (1).

From (1), we have $m_\ell \leq n_\ell$. Since $m_\ell = n$, we conclude that $n_\ell = n$. Since $n_k = n$, this can only mean $\ell = k$. Thus the greedy method is optimal.

Application to Bin Packing. Thus linear bin packing can be solved in $O(n)$ time; if the weights are arbitrary real numbers, this $O(n)$ bound is based on the real RAM computational model of Chapter 1. We can use this solution as a subroutine in solving the original bin packing problem: we just cycle through each of the $n!$ permutations of $w = (w_1, \ldots, w_n)$, and for each compute the greedy solution in $O(n)$ time. The optimal solution is among them. This yields an $\Theta(n \cdot n!) = \Theta((n/e)^{n+(3/2)})$. time algorithm. Here, we assume that we can generate all n-permutations in $O(n!)$ time. This is a nontrivial assumption, but in §6, we will show how this can be done.

We can improve this by a factor of n, since without loss of generality, we may restrict to permutations that begins with an arbitrary w_1 (why?). Since there are $(n-1)!$ such permutations, we obtain:

Lemma 1. *The bin packing problem can be solved in $O(n!) = O((n/e)^{n+(1/2)})$ time in the real RAM model.*

See the Exercise for how this complexity can be improved by another factor of n.

Exercises

Exercise 1.1: Give a counter example to the greedy algorithm for the grouping problem in case the w_i's can be negative. ♢

Exercise 1.2: Suppose the weight w_i's can be negative. How bad can the greedy algorithm be, as a function of the optimal bin size? ♢

Exercise 1.3: There are two places where our optimality proof for the greedy algorithm breaks down when there are negative weights. What are they? ♢

Exercise 1.4: Consider the following "generalized greedy algorithm" in case w_i's can be negative. A solution to linear bin packing be characterized by the indices $0 = n_0 < n_1 < n_2 < \cdots < n_k = n$ where the ith car holds the weights

$$[w_{n_{i-1}+1}, w_{n_i+2}, \ldots, w_{n_i}].$$

Here is a greedy way to define these indices: let n_1 to be the largest index such that $\sum_{j=1}^{n_1} w_j \leq M$. For $i > 1$, define n_i to be the largest index such that $\sum_{j=n_{i-1}+1}^{n_i} w_j \leq M$. Either prove that this solution is optimal, or give a counter example. ♢

Exercise 1.5: Give an $O(n^2)$ algorithm for linear bin packing when there are negative weights. HINT: Use dynamic programming. Assume that when you solve the problem for (M, w), you also solve it for (M, w') where w' is a suffix of w. ♢

Exercise 1.6: Improve the bin packing upper bound in Lemma 1 to $O((n/e)^{n-(1/2)})$. HINT: Repeat the trick which saved us a factor of n in the first place. Fix two weights w_1, w_2. We need to consider two cases: either w_1, w_2 belong to the same bin or they do not. ♢

Exercise 1.7: Two dimensional generalization: suppose that the weights in w are of the form $w_{i,j} = u_i + v_j$ and $(u_1, \ldots, u_m)$ and $(v_1, \ldots, v_n)$ are two given sequences. So w has mn numbers. Moreover, each group must have the form $w(i, i', j, j')$ comprising all $w_{k,\ell}$ such that $i \leq k \leq i'$ and $j \leq \ell \leq j'$. Call this a "rectangular group". We want the sum of the weights in each group to be at most M, the bin capacity. Give a greedy algorithm to form the smallest possible number of rectangular groups. Prove its correctness. ♢

End Exercises

§2. Interval Scheduling Problems

We give more elementary examples of greedy algorithms. An important class of problems involves scheduling intervals.

Typically, we think of an interval I as a time interval, representing some activity. For instance, $I = [s, f)$ where $s < f$ might represent an activity that starts at time s and finishes at time f. Thus I is the set $\{t \in \mathbb{R} : s \leq t < f\}$, usually called a half-open interval. Two activities **conflict** if their time intervals are not disjoint. We use half-open intervals instead of closed intervals so that the finish time of an activity can coincide with the start time of another activitity without causing a conflict. A set $S = \{I_1, \ldots, I_n\}$ of intervals is said to be **compatible** if the intervals in S are pairwise disjoint (i.e., the set is conflict-free).

We begin with the **activities selection problem**, originally studied by Gavril. Imagine you have the choice to do any number of the following fun activities in one afternoon: tennis $1:30 - 3:20$, swimming $1:15 - 2:45$, beach $12:00 - 4:00$, movie $3:00 - 4:30$, movie $4:30 - 6:00$. You are not allowed to do two activities at the same time. Assuming that your goal is to maximize your number of fun activities, which activities should you choose? Formally, the activities selection problem is this: *given a set*

$$A = \{I_1, I_2, \ldots, I_n\}$$

of intervals, to compute a compatible subset of S that is optimal. Here optimality means "of maximum cardinality". E.g., in the above fun activities example, an optimal solution would be to swim and to see two movies. It would be suboptimal to go to the beach. What would a greedy algorithm for this problem look like? Here is a generic version:

```
Generic Greedy Activities Selection:
Input: A a set of intervals
Output: S ⊆ A, a set of compatible intervals
▷ Initialization
    Sort A according to some numerical criterion.
        Let (I_1, ..., I_n) be the sorted sequence.
    Let S = ∅.
▷ Main Loop
    For i = 1 to n
        If S ∪ {I_i} is compatible, add I_i to S
    return(S)
```

Thus, S is a partial solution that we are building up. At stage i, we consider A_i, to either **accept** or **reject** it. Accepting means to make it part of current solution S. But what greedy criteria should we use for sorting? Here are some suggestions:

- Sort I_i's in order of non-decreasing finish times.
- Sort I_i's in order of non-decreasing start times.
- Sort I_i's in order of non-decreasing size $f_i - s_i$.
- Sort I_i's in order of non-decreasing conflict degree. The conflict degree of I_i is the number of I_j's which conflict with I_i.

We now show that the first criterion (sorting by non-decreasing finish times) leads to an optimal solution. In the Exercise, we ask you to show show that all the other criteria do not guarantee optimality.

We use an inductive proof, reminiscent of the joy ride proof. Let $S = (I_1, I_2, \ldots, I_k)$ be the solution given by our greedy algorithm. If $I_i = [s_i, f_i)$, we may assume

$$f_1 < f_2 < \cdots < f_k.$$

Suppose $S' = (I'_1, I'_2, \ldots, I'_\ell)$ is an optimal solution where $I'_i = [s'_i, f'_i)$ and again $f'_1 < f'_2 < \cdots < f'_k$. By optimality of S', we have $k \leq \ell$. CLAIM: We have the inequality $f_i \leq f'_i$ for all $i = 1, \ldots, k$. We leave this proof as an exercise.

Let us now derive a contradiction if the greedy solution is not optimal: assume $k < \ell$ so that I'_{k+1} is defined. Then

$$\begin{array}{rcll} f_k & \leq & f'_k & (\text{ by } CLAIM) \\ & \leq & s'_{k+1} (\text{since } I'_k, I'_{k+1} \text{ have no conflict}) & \end{array}$$

and so I'_{k+1} is compatible with $\{I_1, \ldots, I_k\}$. This is a contradition since the greedy algorithm halts after choosing I_k because there are no other compatible intervals.

What is the running time of this algorithm? In deciding if interval I_i is compatible with the current set S, it is enough to only look at the finish time f of the last accepted interval. This can be done in $O(1)$ time since this comparison takes $O(1)$ and f can be maintained in $O(1)$ time. Hence the algorithm takes linear time after the initial sorting.

Extensions, variations. There are many possible variations and generalizations of the activities selection problem. Some of these problems are explored in the Exercises.

- Suppose your objective is not to maximize the number of activities, but to maximize the total amount of time spent in doing activities. In that case, for our fun afternoon example, you should go to the beach and see the second movie.
- Suppose we generalize the objective function by adding a weight ("pleasure index") to each activity. Your goal now is to maximize the total weight of the activities in the compatible set.
- We can think of the activities to be selected as a uni-processor scheduling problem. (You happen to be the processor.) We can ask: what if you want to process as many activities as possible using two processors? Does our original greedy approach extend in the obvious way? (Find the greedy solution for processor 1, then find greedy solution for processor 2).
- Alternatively, suppose we ask: what is the minimum number of processors that suffices to do all the activities in the input set?
- Suppose that, in addition to the set A of activities, we have a set C of classrooms. We are given a bipartite graph with vertices $A \cup C$ and edges is $E \subseteq A \times C$. Intuitively, $(I, c) \in E$ means that activity I can be held in classroom c. We want to know whether there is an assignment $f : A \to C$ such that (1) $f(I) = c$ implies $(I, c) \in E$ and (2) $f^{-1}(c)$ is compatible. REMARK: scheduling of classrooms in a school is more complicated in many more ways. One complication is the need to do weekly scheduling, not daily scheduling.

Exercises

Exercise 2.1: We gave four different greedy criteria for the activities selection problem.
(a) Show that the other three criteria are suboptimal.
(b) Actually, each of the four criteria has a inverted version, where we sort in non-increasing order. Show that each of these inverted criteria are also suboptimal. ♢

Exercise 2.2: Suppose the input $A = (I_1, \ldots, I_n)$ for the activities selection problem is already sorted, by non-decreasing order of their start times, i.e., $s_1 \leq s_2 \leq \cdots \leq s_n$. Give an algorithm to compute a optimal solution in $O(n)$ time. Show that your algorithm is correct. ♢

Exercise 2.3: Again consider the activities selection problem. We now want to maximize the total **length** of all the activities in a $S \subseteq A$. Here, the length of an activity $I = [s, f)$ is just $f - s$. In case S is not compatible, we define its length to be 0. Let $A_{i,j} = \{I_i, I_{i+1}, \ldots, I_j\}$ for $i \leq j$ and $F_{i,j}$ be an optimal solution for $A_{i,j}$.
(a) Show by a counter-example that the following "dynamic programming principle" fails:

$$F_{i,j} = \overline{\max}_{i \leq k \leq j-1} F_{i,k} \cup F_{k+1,j}$$

where $\overline{\max}\{F_1, F_2, \ldots, F_m\}$ returns the set F_ℓ whose length is maximum. (Recall that the length of F_ℓ is zero if it is not feasible.
(b) Give an $O(n \log n)$ algorithm for this problem. HINT: order the activities in the set S according to their finish times, say,

$$f_1 \leq f_2 \leq \cdots \leq f_n.$$

Consider the set of subproblems $S_i := \{I_1, \ldots, I_i\}$ for $i = 1, \ldots, n$. Use an incremental algorithm to solve $S_1, S_2, \ldots, S_n$ in this order. ♢

Exercise 2.4: Give a divide-and-conquer algorithm for the problem in previous exercise, to find the maximum length feasible solution for a set S of activities. (This approach is harder and less efficient!) ♢

End Exercises

§3. Huffman Code

We begin with an informally stated problem:

(P) Given a string s of characters (or letters or symbols) taken from an alphabet Σ, choose a *variable length code* C for Σ so as to minimize the space to encode the string s.

Before making this problem precise, it is helpful to know the context of such a problem. A computer file may be regarded as a string s, so problem (P) can be called the **file compression problem**. Typically, characters in computer files are encoded by a *fixed-length binary code* (usually the ASCII standard). Note that in this case, each code word has length at least $\log_2 |\Sigma|$. The idea of using variable length code is to take advantage of the relative frequency of different characters. For instance, in typical English texts, the letters 'e' and 't' are most frequent and it is a good idea to use shorter length codes for them. An example of a variable length code is Morse code (see Notes at the end of this section).

A (binary) **code** for Σ is an injective function

$$C : \Sigma \to \{0,1\}^*.$$

A string of the form $C(x)$ ($x \in \Sigma$) is called a **code word**. The string $s = x_1 x_2 \cdots x_m \in \Sigma^*$ is then encoded as

$$C(s) := C(x_1)C(x_2)\cdots C(x_m) \in \{0,1\}^*.$$

This raises the problem of decoding $C(s)$, *i.e.*, recovering s from $C(s)$. For a general C and s, one cannot expect unique decoding. One solution is to introduce a new symbol '\$' and use it to separate each $C(x_i)$. If we insist on using binary alphabet for the code, this forces us to convert, say, '0' to '00', '1' to '01' and '\$' to '11'. This doubles the number of bits, and seems to be wasteful.

Prefix-free codes. The standard solution for unique decoding is to insist that C be **prefix-free**. This means that if $a, b \in \Sigma$, $a \neq b$, then $C(a)$ is not a prefix of $C(b)$. It is not hard to see that the decoding problem is uniquely defined for prefix-free codes. With suitable preprocessing (basically to construct the "code tree" for C, defined next) decoding can be done very simply in an on-line fashion. We leave this for an exercise.

We represent a prefix-free code C by a binary tree T_C with $n = |\Sigma|$ leaves. Each leaf in T_C is labeled by a character $b \in \Sigma$ such that the path from the root to b is represented by $C(b)$ in the natural way: starting from the root, we use successive bits in $C(b)$ to decide to make a left branch or right branch from the current node of T_C. We call T_C a **code tree** for C. Figure 1 shows two such trees representing prefix codes for the alphabet $\Sigma = \{a, b, c, d\}$. The first code, for instance, corresponds to $C(a) = 00$, $C(b) = 010$, $C(c) = 011$ and $C(d) = 1$.

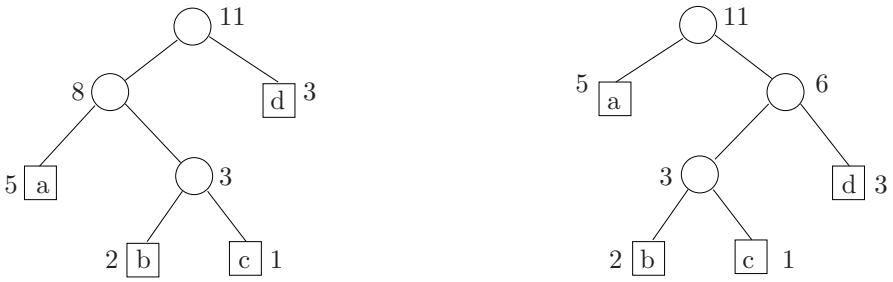


Figure 1: Two prefix-free codes and their code trees: assume $f(a) = 5, f(b) = 2, f(c) = 1, f(d) = 3$.

Returning to the informal problem (P), we can now interpret this problem as the construction of the best prefix-free code C for s, *i.e.*, the code that minimizes the length of $C(s)$. It is easily seen that the only statistics important about s is its frequency function f_s where $f_s(x)$ is the number of occurrences of the character x in s. In general, call a function of the form

$$f : \Sigma \to \mathbb{N}$$

a **frequency function**. So we now regard the input data to our problem as a frequency function $f = f_s$ rather than a string s. Relative to f, the **cost** of C will be defined to be

$$COST(f, C) := \sum_{a \in \Sigma} |C(a)| \cdot f(a). \tag{2}$$

Clearly $COST(f_s, C)$ is the length of $C(s)$. Finally, the **cost** of f is defined to be

$$COST(f) := \min_C COST(f, C)$$

over all prefix-free codes C on the alphabet Σ. A code C is **optimal** for f if $COST(f, C)$ attains this minimum. It is easy to see that an optimal code tree must be a *full* binary tree (i.e., non-leaves must have two children).

For the codes in Figure 1, assuming the frequencies of the characters a, b, c, d are $5, 2, 1, 3$ (respectively), the cost of the first code is $5 \cdot 2 + 2 \cdot 3 + 1 \cdot 3 + 3 \cdot 1 = 22$. The second code is better, with cost 20.

We now precisely state the informal problem (P) as the **Huffman coding problem**:

Given a frequency function $f : \Sigma \to \mathbb{N}$, find an optimal prefix-free code C for f.

Relative to a frequency function f on Σ, we associate a **weight** $W(u)$ with each node u of the code tree T_C: the weight of a leaf is just the frequency $f(x)$ of the character x at that leaf, and the weight of an internal node is the sum of the weights of its children. Let $T_{f,C}$ denote such a **weighted code tree**. For example, see Figure 1 where the frequency of each node is written next to it. The **frequency** of $T_{f,C}$ is the frequency of its root, and its **cost** $COST(T_{f,C})$ is the sum of the frequencies of all its *internal* nodes. In Figure 1(a), the internal nodes have frequencies $3, 8, 11$ and so the $COST(T_{f,C}) = 3 + 8 + 11 = 22$. In general, the reader may verify that

$$COST(f, C) = COST(T_{f,C}). \tag{3}$$

We need the **merge** operation on code trees: if T_i is a code tree on the alphabet Σ_i $(i = 1, 2)$ and $\Sigma_1 \cap \Sigma_2$ is empty, then we can merge them into a code tree T on the alphabet $\Sigma_1 \cup \Sigma_2$ by introducing a new node as the root of T and T_1, T_2 as the two children of the root. We also write $T_1 + T_2$ for T. If T_1, T_2 are weighted code trees, the result T is also a weighted code tree.

We now present a greedy algorithm for the Huffman coding problem:

Huffman Code Algorithm:
Input: frequency function $f : \Sigma \to \mathbb{N}$.
Output: optimal code tree T^* for f.
1. Let S be a set of weighted code trees. Initially, S is the set of $n = |\Sigma|$ trivial trees, each tree having only one node representing a single character in Σ.
2. while S has more than one tree,
   2.1. Choose $T, T' \in S$ with the minimum and the next-to-minimum frequencies, respectively.
   2.2. Merge T, T' and insert the result $T + T'$ into S.
   2.3. Delete T, T' from S.
3. Now S has only one tree T^*. Output T^*.

Let us illustrate the algorithm with perhaps the most famous 12-letter string in computing: `hello world!`. The alphabet Σ for this string and its frequency function may be represented by the following two arrays:

letter	h	e	l	o	⊔	w	r	d	!
frequency	1	1	3	2	1	1	1	1	1

Note that the exclamation mark (!) and blank space (⊔) are counted as letters in the alphabet Σ. The final Huffman tree is shown in Figure 2. The number shown inside a node u of the tree is the **weight** of the node.

This is just sum of the frequencies of the leaves in the subtree at u. Each leaf of the Huffman tree is labeled with a letter from Σ.

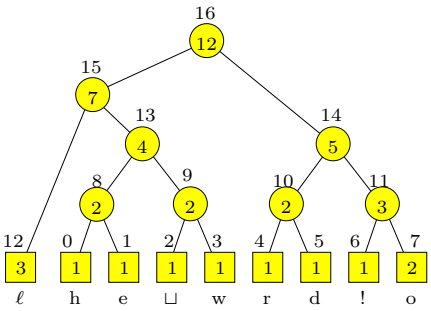


Figure 2: Huffman Tree for `hello world!`

To trace the execution of our algorithm, in Figure 2 we indicate the order $(0, 1, 2, \ldots, 16)$ in which the nodes were placed into the priority queue. For instance, the leaf `h` is the first to be placed in the queue, and the root is the last (16th) to be placed in the queue.

Implementation and complexity. The input for the Huffman algorithm may be implemented as an array $f[1..n]$ where $f[i]$ is the frequency of the ith letter and $|\Sigma| = n$. The output is a binary tree whose leaves are labeled from 1 to n. This algorithm can be implemented using a priority queue on a set S of binary tree nodes. Recall (§III.2) that a priority queue supports two operations, (a) inserting a keyed item and (b) deleting the item with smallest key. The frequency of the code tree serves as its key. Any balanced binary tree scheme (such as AVL trees in Lecture IV) will give an implementation in which each queue operation takes $O(\log n)$ time. Hence the overall algorithm takes $O(n \log n)$.

Correctness. We show that the produced code C has minimum cost. This depends on the following simple lemma. Let us say that a pair of nodes in T_C is a **deepest pair** if they are siblings and their depth is the depth of the tree T_C. In a full binary tree, there is always a deepest pair.

Lemma 2. *There is an optimal Huffman code in which the two least frequent characters forms a deepest pair.*

Proof. Suppose b, c are two characters at depths $D(b), D(c)$ (respectively) in a weighted code tree T. If we exchange the weights of these two nodes to get a new code tree T' where

$$\begin{aligned} COST(T) - COST(T') &= f(b)D(b) + f(c)D(c) - f(b)D(c) - f(c)D(b) \\ &= [f(b) - f(c)][D(b) - D(c)] \end{aligned}$$

where f is the frequency function. If b has the least frequency and $D(c)$ is the depth of the tree T then clearly

$$COST(T) - COST(T') \geq 0.$$

Hence if c, c' are the two characters labeling a deepest pair and and b, b' are the two least frequent characters, then by a similar argument, we may exchange the labels $b \leftrightarrow b'$ and $c \leftrightarrow c'$ without increasing the cost of the code. **Q.E.D.**

We are ready to prove the correctness of Huffman's algorithm. Suppose by induction hypothesis that our algorithm produces an optimal code whenever the alphabet size $|\Sigma|$ is less than n. The basis case, $n = 1$, is trivial. Now suppose $|\Sigma| = n > 1$. After the first step of the algorithm in which we merge the two least

frequent characters b, b', we can regard the algorithm as constructing a code for a modified alphabet Σ' in which b, b' are replaced by a new character $[bb']$ with modified frequency f' such that $f'([bb']) = f(b) + f(b')$, and $f'(x) = f(x)$ otherwise. By induction hypothesis, the algorithm produces the optimal code C' for f':

$$COST(f') = COST(f', C'). \tag{4}$$

This code C' is related to a suitable code C for Σ in the obvious way and satisfies

$$COST(f, C) = COST(f', C') + f(b) + f(b'). \tag{5}$$

It is easily seen from our lemma that

$$COST(f) = COST(f') + f(b) + f(b'). \tag{6}$$

From equations (4), (5) and (6), we conclude $COST(f) = COST(f, C)$, *i.e.*, C is optimal. ■

Remarks: The publication of this algorithm in 1952 by D. A. Huffman was considered a major achievement. This algorithm is clearly useful for compressing binary files. See "Conditions for optimality of the Huffman Algorithm", D.S. Parker (*SIAM J.Comp.*, 9:3(1980)470–489, *Erratum* 27:1(1998)317), for a variant notion of cost of a Huffman tree and characterizations of the cost functions for which the Huffman algorithm remains valid.

Notes on Morse Code. In the Morse[1] code, letters are represented by a sequence of dots and dashes: $a = \cdot -$, $e = \cdot$, $t = -$ and $z = - - \cdot\cdot$. The code is also meant to be sounded: dot is pronounced '*dit*' (or '*di-*' when non-terminal), dash is pronounced '*dah*' (or '*da-*' when non-terminal). Thus '*a*' is $di - dah$, '*z*' is $da - da - di - dit$. Clearly, Morse code is not prefix-free. It also is no capital or small letters. Here is the full alphabet:

Letter	Code	Letter	Code
A	· –	B	– · · ·
C	– · – ·	D	– · ·
E	·	F	· · – ·
G	– – ·	H	· · · ·
I	· ·	J	· – – –
K	– · –	L	· – · ·
M	– –	N	– ·
O	– – –	P	· – – ·
Q	– – · –	R	· – ·
S	· · ·	T	–
U	· · –	V	· · · –
W	· – –	X	– · · –
Y	– · – –	Z	– – · ·
0	– – – – –	1	· – – – –
2	· · – – –	3	· · · – –
4	· · · · –	5	· · · · ·
6	– · · · ·	7	– – · · ·
8	– – – · ·	9	– – – – ·
Fullstop (.)	· – · – · –	Comma (,)	– – · · – –
Query (?)	· · – – · ·	Slash (/)	– · · – ·
BT (pause)	– · · · –	AR (end message)	· – · – ·
SK (end contact)	· · · – · –		

[1]Samuel Finley Breese Morse (1791-1872) was Professor of the Literature of the Arts of Design in the University of the City of New York (now New York University) 1832-72. It was in the university building on Washington Square where he completed his experiments on the telegraph.

How do you send messages using Morse code? Note that spaces are not part of the Morse alphabet! Since space are important in practice, it has an informal status as an explicit character (so Morse code is not strictly a binary code). There are 3 kinds of spaces: space between *dit*'s and *dah*'s within a letter, space between letters, and space between words. Let us assume some concept of **unit space**. Then the above three types of spaces are worth 1, 3 and 7 units, respectively. These units can also be interpreted as "unit time" when the code is sounded. Hence we simply say **unit** without prejudice. Next, the system of dots and dashes can also be brought into this system. We say that spaces are just "empty units", while *dit*'s and *dah*'s are "filled units". *dit* is one filled unit, and *dah* is 3 filled units. Of course, this brings in the question: why 3 and 7 instead of 2 and 4 in the above?

Exercises

Exercise 3.1: Give an optimal Huffman code for the frequencies of the letters of the alphabet:

$$a = 5, b = 1, c = 3, d = 3, e = 7, f = 0, g = 2, h = 1, i = 5, j = 0, k = 1, l = 2, m = 0,$$
$$n = 5, o = 3, p = 0, q = 0, r = 6, s = 3, t = 4, u = 1, v = 0, w = 0, x = 0, y = 1, z = 1.$$

Please determine the cost of the optimal tree and show your intermediate collections of code trees. NOTE: you need not to give any code word to symbols with the zero frequency. ♢

Exercise 3.2: What is the *length* of the Huffman code for the following string $s =$ "`please compress me`". The length of s is 18. Show your hand computation. ♢

Exercise 3.3: (a) Prove (3).
(b) It is important to note that we defined $COST(T_{f,C})$ to be the sum of $f(u)$ where u range over the *internal* nodes of $T_{f,C}$. That means that if $|\Sigma| = 1$ (or $T_{f,C}$ has only one node which is also the root) then $COST(T_{f,C}) = 0$. Why does Huffman code theory break down at this point?
(c) Suppose we (accidentally) defined $COST(T_{f,C})$ to be the sum of $f(u)$ where u range over the *all* nodes of $T_{f,C}$. Where in your proof in (a) would the argument fail? ♢

Exercise 3.4: Below is President Lincoln's address at Gettysburg, Pennsylvania on November 19, 1863.
(a) Give the Huffman code for the string S comprising the first two sentences of the address. Also state the length of the Huffman code for S, and the percentage of compression so obtained (assume that the original string uses 7 bits per character). You need to distinguish caps and small letters, introduce symbols for space and punctuation marks. But ignore the newline characters.
(b) The previous part was meant to be done by hand. Now write a program in your favorite programming language to compute the Huffman code for the entire Gettysburg address. What is the compression obtained?

```
Four score and seven years ago our fathers brought forth on this
continent a new nation, conceived in liberty and dedicated to the
proposition that all men are created equal.  Now we are engaged in a
great civil war, testing whether that nation or any nation so conceived
and so dedicated can long endure. We are met on a great battlefield of
that war. We have come to dedicate a portion of that field as a final
resting-place for those who here gave their lives that that nation
might live. It is altogether fitting and proper that we should do this.
But in a larger sense, we cannot dedicate, we cannot consecrate, we
cannot hallow this ground.  The brave men, living and dead who
```

```
struggled here have consecrated it far above our poor power to add or
detract. The world will little note nor long remember what we say here,
but it can never forget what they did here.  It is for us the living
rather to be dedicated here to the unfinished work which they who
fought here have thus far so nobly advanced. It is rather for us to be
here dedicated to the great task remaining before us--that from these
honored dead we take increased devotion to that cause for which they
gave the last full measure of devotion--that we here highly resolve
that these dead shall not have died in vain, that this nation under God
shall have a new birth of freedom, and that government of the people,
by the people, for the people shall not perish from the earth.
```

♢

Exercise 3.5: Let $(f_0, f_1, \ldots, f_n)$ be the frequencies of $n+1$ symbols (assuming $|\Sigma| = n+1$). Consider the Huffman code in which the symbol with frequency f_i is represented by the ith code word in the following sequence

$$1, 01, 001, 0001, \ldots, \underbrace{00\cdots01}_{n-1}, \underbrace{00\cdots001}_{n}, \underbrace{00\cdots000}_{n}.$$

(a) Show that a sufficient condition for optimality of this code is

$$\begin{aligned} f_0 &\geq f_1 + f_2 + f_3 + \cdots + f_n, \\ f_1 &\geq f_2 + f_3 + \cdots + f_n, \\ f_2 &\geq f_3 + \cdots + f_n, \\ &\cdots \\ f_{n-2} &\geq f_{n-1} + f_n. \end{aligned}$$

(b) Suppose the frequencies are distinct. Give a set of sufficient and necessary conditions. ♢

Exercise 3.6: Suppose you are given the frequencies f_i in sorted order. Show that you can construct the Huffman tree in linear time. ♢

Exercise 3.7: Suppose our alphabet is the set $\Sigma = \{0, \ldots, n-1\}$. Each $a \in \Sigma$ is really a binary string of length $\lceil \lg n \rceil$. Let T be any Huffmann code tree for Σ. Show how we can represent T using at most $2n - 1 + n\lceil \lg n \rceil$ bits. To understand what is needed, suppose $r(T) \in \{0,1\}$ is the representation of T. Suppose I have a message $M \in \Sigma^*$ and it is encoded as $c(M) \in \{0,1\}^*$ uing the code of T. You must do 3 things:
(a) Describe $r(T)$ for a Huffmann code tree T for $\{0, \ldots, n-1\}$.
(b) If T is the second tree in figure 1, and assuming $a = 3, b = 0, c = 2, d = 1$, what is $r(T)$?
(c) Describe how to reconstruct T from $r(T)$.
HINT: encode the full binary tree by a systematic traversal of all the nodes, level by level. ♢

Exercise 3.8: Generalize to 3-ary Huffman codes, $C : \Sigma \to \{0,1,2\}^*$, represented by the corresponding 3-ary code trees (where each node has degree at most 3):
(a) Show that in an optimal 3-ary code tree, any node of degree 2 must have leaves as both its children.
(b) Show that there are either no degree 2 nodes (if $|\Sigma|$ is odd) or one degree 2 node (if $|\Sigma|$ is even).
(c) Show that when there is one degree 2 node, then the depth of its children must be the height of the tree.
(d) Give an algorithm for constructing an optimal 3-ary code tree and prove its correctness. ♢

Exercise 3.9: Further the above 3-ary Huffman tree construction to arbitrary k-ary codes for $k \geq 4$. ♢

Exercise 3.10: Suppose that the cost of a binary code word w is $z + 2o$ where z (resp. o) is the number of zeros (resp. ones) in w. Call this the **skew cost**. So ones are twice as expensive as zeros (this cost model might be realistic if a code word is converted into a sequence of dots and dashes as in Morse code). We extend this definition to the **skew cost** of a code C or of a code tree. A code or code tree is **skew Huffman** if it is optimum with respect to this skew cost. For example, see figure 3 for a skew Huffman tree for alphabet $\{a, b, c\}$ and $f(a) = 3$, $f(b) = 1$ and $f(c) = 6$.

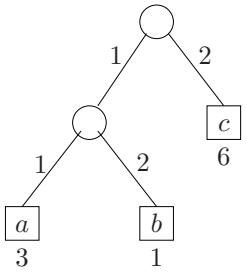


Figure 3: A skew Huffman tree with skew cost of 21.

(a) Argue that in some sense, there is no greedy solution that makes its greedy decisions based on a linear ordering of the frequencies.
(b) Consider the special case where all letters of the alphabet has equal frequencies. Describe the shape of such code trees. For any n, is the skew Huffman tree unique?
(c) Give an algorithm for the special case considered in (b). Be sure to argue its correctness and analyze its complexity. HINT: use an "incremental algorithm" in which you extend the solution for n letters to one for $n + 1$ letters. ♢

Exercise 3.11: (Golin-Rote) Further generalize the problem in the previous exercise. Fix $0 < \alpha < \beta$ and let the cost of a code word w be $\alpha \cdot z + \beta \cdot o$. Suppose α/β is a rational number. Show a dynamic programming method that takes $O(n^{\beta+2})$ time. NOTE: The best result currently known gets rid of the "+2" in the exponent, at the cost of two non-trivial ideas. ♢

Exercise 3.12: (Open) Give a non-trivial algorithm for the problem in the previous exercise where α/β is not rational. An algorithm is "trivial" here if it essentially checks all binary trees with n leaves. ♢

Exercise 3.13: Suppose that the "frequency" of a symbol can be negative (this is really an abuse of the term frequency). But we can define the cost of an optimal code tree as before. Is the greedy solution still optimal? ♢

Exercise 3.14: (Elias) Consider the following binary encoding scheme for the infinite alphabet $\mathbb{N}$ (the natural numbers): an integer $n \in \mathbb{N}$ is represented by a prefix string of $\lfloor \lg n \rfloor$ 0's followed by the binary representation of n. This requires $1 + 2\lfloor \lg n \rfloor$ bits.
(a) Show that this is a prefix-free code.
(b) Now improve the above code as follows: replacing the prefix of $\lfloor \lg n \rfloor$ 0's and the first 1 by a representation of $\lfloor \lg n \rfloor$ the same scheme as (a). Now we use only $1 + \lfloor \lg n \rfloor + 2\lfloor \lg(1 + \lg n) \rfloor$ bits to encode n. Again show that this is a prefix-free code. ♢

Exercise 3.15: (Shift Key in Huffman Code) We want to encode small as well as capital letters in our alphabet. Thus 'a' and 'A' are to be distinguished. There are two ways to achieve this: (I) View the small and capital letters as distinct symbols. (II) Introduce a special "shift" symbol, and each letter is assumed to be small unless it is preceded by a shift symbol, in which case it is considered a capital. Use the text of this question as your input string. Punctuation marks and spaces are part of this string. But new lines (CRLF) do not contribute any symbols to the string. But when you merge two lines, sometimes the CRLF character is sometimes replaced by a space. Also, in standard typography, the space between two sentences is a double space. For our purposes, assume all spaces are single space.
(a) Compute the Huffman code tree for coding the above string using method (I). Note that the string begins with the words "We want to en..." and ends with "...ces are single space.". Be sure to compute the number of bits in the Huffman code for this string.
(b) Same as part (a) but using method (II).
(c) Discuss the pros and cons of (I) and (II).
(d) There are clearly many generalizations of shift keys, as seen in modern computer keyboards. The general problem arises when our letters or characters are no longer indivisible units, but exhibit structure (as in Chinese characters). Give a general formulation of such extensions. ♢

End Exercises

§4. Dynamic Huffman Code

Here is the typical sequence of steps for compressing and transmitting a string s using the Huffman code algorithm:

(i) First make a pass over the string s to compute its frequency function.

(ii) Next compute a Huffman code tree T_C corresponding to some code C.

(iii) Using T_C, compute the compressed string $C(s)$.

(iv) Finally, transmit the tree T_C, together with the compressed string $C(s)$, to the receiver.

The receiver receives T_C and $C(s)$, and hence can recover the string s. Since the sender must process the string s in two passes (steps (i) and (iii)), the original Huffman tree algorithm is sometimes called the "2-pass Huffman encoding algorithm". There are two deficiencies with this 2-pass process: (a) Multiple passes over the input string s makes the algorithm unsuitable for realtime data transmissions. Note that if s is a large file, this require extra buffer space. (b) The Huffman code tree must be explicitly transmitted before the decoding can begin. We need some way to encode T_C. This calls for a separate algorithm to handle T_C in the encoding and decoding process.

An approach called "Dynamic Huffman coding" (or adaptive Huffman coding) overcomes these problems: it passes over the string s only once, and there is no need to explicitly transmit the code tree. Two known algorithms for dynamic Huffman coding [6] are the **FGK Algorithm** (Faller 1973, Gallager 1978, Knuth 1985) and the **Lambda Algorithm** (Vitter 1987). Vitter's algorithm ensures that the transmitted string length is $\leq H_2(s) + |s| - 1$ where $H_2(s)$ is the number of bits transmitted by the 2-pass algorithm for s, independent of alphabet size. It can be shown that the FGK Algorithm transmit at most $2H_2(s) + 4|s|$ bits.

The key idea here is the Sibling Property of Gallagher. Let T be a full binary tree on $k \geq 0$ internal nodes. Suppose there is a non-negative integer weight on each node such that the weight of an internal node

is the sum of the weights of its two children. Recall that "full" means each internal node of T has exactly two children. It is easy to see that T has $k+1$ leaves or $2k+1$ nodes in all. Call such a tree T a **pre-Huffman tree**. We say T has the **Sibling Property** if its nodes can be **ranked** from 0 to $2k$ satisfying:

(S1) If w_i is the weight of node with rank i, then $w_{i-1} \leq w_i$ for $i = 1, \ldots, 2k$.

(S2) The nodes with ranks $2j$ and $2j+1$ are siblings (for $j = 0, \ldots, k-1$).

For example, the nodes of the pre-Huffmann tree in Figure 2 has been given the rankings $0, 1, 2, \ldots, 16$. We check that this ranking satisfies the Sibling Property.

Note that node with rank $2k$ is necessarily the root, and it has no siblings. In general, let $r(u)$ denote the rank of node u. If the weights of nodes are all distinct, then the rank $r(u)$ is uniquely determined by Property (S1).

Let T be a weighted code tree. Then T is clearly pre-Huffman. We say T is **Huffman** if T can be the output of the Huffman code algorithm. In this definition, we view the Huffman code algorithm as a nondeterministic process: *when two or more nodes have equal weights, the choice of the next two nodes for merging is regarded as a nondeterministic choice.* Hence there can be many potential output trees.

Lemma 3. *Let T be pre-Huffman. Then T is Huffman iff it has the Sibling Property.*

Proof. If T is Huffman then we can rank the nodes in the order that two nodes are merged, and this ordering implies the Sibling Property. Conversely, the Sibling Property determines an obvious order for merging pairs of nodes to form a Huffman tree. **Q.E.D.**

Example: the code tree in Figure 2 is Huffman. Since it is Huffman, there must be some ranking that satisfies the Sibling Property. Indeed, such a ranking has already been indicated.

The Restoration Problem. The key problem of dynamic Huffman tree is how to restore Huffman-ness under a particular kind of perturbation: let T be Huffman and suppose the weight of a leaf u is incremented by 1. So weights of all the nodes along the path from u to the root are similarly incremented. The result is a pre-Huffman tree T', but it may not be Huffman any more. *The problem is to restore Huffman-ness in such a tree T'.*

Consider the following algorithm for restoring Huffman-ness in T. For each node v in T, let $W[v]$ and $R[v]$ denote the weight and rank of v in the original tree T that satisfies the Sibling Property is satisfied. Let u be the current node. Initially, u is the leaf whose weight was incremented. We use the following iterative process:

```
Restore (u)
    While u is not the root do
        Find the node v with the largest rank R[v]
            subject to the constraint W[v] = W[u].
        If v = u then W[u]++ and let u = parent(u). Break.
        If v ≠ u then swap u and v.
                ◁ This really swaps the entire subtree at u and v.
            Increment W[u]++
            Let u = parent(u). Break.
                ◁ Note that u is now the former parent of v.
    Increment W[u]++.    ◁ u is the root
```

Swapping needs to be explained: imagine the nodes of T are stored in a linear list

$$(u_1, u_2, \ldots, u_n)$$

in the order of their ranks: $R[u_i] = i$. Swapping u and v means that their ranks are exchanged; so their positions in this list are swapped. Thus the rank of the current node is strictly increased by such swaps. Each node u has three pointers, $u.\texttt{left}, u.\texttt{right}$ and $u.\texttt{parent}$. When we swap u and v, their siblings may have changed (recall that rank $2j$ and rank $2j+1$ nodes must be siblings). So the child pointers of $u.\texttt{parent}$ and $v.\texttt{parent}$ must be changed appropriately.

Let us consider an example to see how Restore works, using the famous string `hello world!`. Suppose we have just completely processed this string, and the current Huffman tree T is given in Figure 2. Let the next character to be transmitted be $\sqcup$. Let u be the node corresponding to $\sqcup$. So $W[u]$ is incremented. But this may now destroy the Sibling Property. But if u is make the rank ...

Our dynamic Huffman code tree T must be capable to expanding its alphabet. E.g., if the current alphabet is $\Sigma = \{\texttt{h}, \texttt{e}\}$ and we next encounter the letter `l`, we want to expand the alphabet to $\Sigma = \{\texttt{h}, \texttt{e}, \texttt{l}\}$. For this purpose, we introduce in T a special leaf with weight 0. Call this the 0**-node**. This node does not represent any letters of the alphabet, but in another sense, it represents all the yet unseen letters. We might say that the 0-node represents the character '$*$'. Upon seeing a new letter like `l`, we "expand" the 0-node so that its left child is the new 0-node, and its right child u is a new leaf representing the letter `l`. The frequency and weight of u is 1, so the original 0-node now has weight 1. We must now call Restore on the parent of the original 0-node.

Here now is the dynamic Huffman coding method for transmitting a string s:

Dynamic Huffman Transmission Algorithm:
Input: a string s of length n.
Output: the dynamically encoded sequence representing s.
▷ *Initialization*
Initialize T to contain just the 0-node.
▷ *Main Loop*
for $i = 1$ to n do
1. Let x be the ith character in the input string s.
2. If x is stored in node u in the current tree T,
3. transmit the current code word for u, and
5. call Restore(u).
6. Else ◁ *x is a new character*
7. transmit the code word for the current 0-node;
8. transmit the canonical representation for x;
◁ *E.g., the ASCII code for x*
9. expand the 0-node to have two children, both with weight 0;
10. let the right sibling u represent the character x
11. and the left sibling represent the new 0-node.
12. Call Restore(u).

Decoding is also relatively straightforward. We are processing a continuous binary sequence, but we know where the implicit "breaks" are in this continuous sequence. Call the binary sequence between these breaks a **word**. We know how to recognize these words by maintaining the same dynamic Huffman code tree T as the transmission algorithm. For each received word, we know whether it is (a) a code word for some character, (b) signal to add a new letter to the alphabet Σ, or (c) the canonical representation of a letter. Using this information, we can update T and also produce the next character in the string s.

Compact Representation of Huffman Tree. In our Huffman tree algorithm, we represented the Huffman tree as a binary tree. We now consider a more compact representation of a Huffman tree T by exploiting its Sibling property: suppose T has $k \geq 1$ leaves. Each of its $2k-1$ nodes is identified by its rank, *i.e.*, a number from 0 to $2k-2$. Hence node i has rank i. We use two arrays

$$W[0..2k-2], \qquad L[0..2k-2]$$

of length $2k-1$ where $W[i]$ is the weight of node i, and $L[i]$ is the left child of node i. So $L[i]+1$ is the right child of node i. In case node i is a leaf, we let $L[i] = -1$.

Another issue which we side-stepped until now is the practical but indispensable step of mapping between a letter $x \in \Sigma$ and the Huffman code of the letter x. We shall view Σ as a subset of a fixed universal set U where $U \subseteq \{0,1\}^N$. The elements of U may be called the **canonical code**. In reality, U might be the set of ASCII characters and $N = 7$. We assume the transmitter and receiver both know this global parameter N and the set U. For instance, U may be the set of ASCII characters and so $N = 7$. To map leaf i to its canonical code, we introduce a third array

$$C[1..k]$$

where $C[i] \in U$.

Another approach to dynamic compression of strings is based on the move-to-front heuristic and splay trees [1].

Exercises

Exercise 4.1: Give a careful and efficient implementation of the dynamic Huffman code. Assume the compact representation of Huffman tree using the arrays W and L described in the text. ♢

Exercise 4.2: A previous exercise (1.2) asks you to construct the standard Huffman code of Lincoln's speech at Gettysburg.
(a) Construct the optimal Huffman code tree for this speech. Please give the length of Lincoln's coded speech. Also give the size of the code tree (use Exercise 1.5).
(b) Please give the length of the dynamic Huffman code for this speech. How much improvement is it over part (a)? Also, what is the code tree at the end of the dynamic coding process? ♢

Exercise 4.3: The correctness of the dynamic Huffman code depends on the fact that the weight at the leaves are integral and the change is $+1$.
(a) Suppose the leave weights can be any real number, and the change in weight is also an arbitrary positive number. Modify the algorithm.
(b) What if the weight change can be negative? ♢

Exercise 4.4: Consider 3-ary Huffman tree code. State and prove the Sibling property for this code. ♢

End Exercises

§5. Matroids

An abstract structure that supports greedy algorithms is matroids. We first illustrate the concept.

Graphic matroids. Let $G = (V, S)$ be a bigraph. A subset $A \subseteq S$ is **acyclic** if it does not contain any cycle. Let I be the set of all acyclic subsets of S. The empty set is a acyclic and hence belongs to I. We note two properties of I:

Hereditary property: If $A \subseteq B$ and $B \in I$ then $A \in I$.

Exchange property: If $A, B \in I$ and $|A| < |B|$ then there is an edge $e \in B - A$ such that $A \cup \{e\} \in I$.

The hereditary property is obvious. To prove the exchange property, note that the subgraph $G_A := (V, A)$ has $|V| - |A|$ (connected) components; similarly the subgraph $G_B := (V, B)$ has $|V| - |B|$ components. If every component $U \subseteq V$ of G_B is contained in some component of U' of G_A, then $|V| - |B| < |V| - |A|$ implies that some component of G_A contains no vertices, contradiction. Hence assume $U \subseteq V$ is a component of G_B that is not contained in any component of G_A. Let $T := B \cap \binom{U}{2}$. Thus (U, T) is a tree and there must exist an edge $e = (u{-}v) \in T$ such that u and v belongs to different components of G_A. This e will serve for the exchange property.

For example, in figure 4 the sets $A = \{a{-}b, a{-}c, a{-}d\}$ and $B = \{b{-}c, c{-}a, a{-}d, d{-}e\}$ are acyclic. Then the exchange property is witnessed by the edge $d{-}e$.

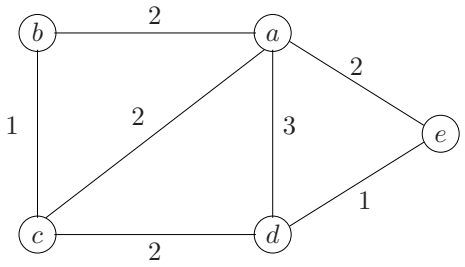


Figure 4: A bigraph with edge costs.

Matroids. The above system (S, I) is called the **graphic matroid** corresponding to graph $G = (V, S)$. In general, a **matroid** is a hypergraph or **set system**

$$M = (S, I)$$

where S is a non-empty set, I is a non-empty family of subsets of S (*i.e.*, $I \subseteq 2^S$) such that I has both the hereditary and exchange properties. The set S is called the **ground set**. Elements of I are called **independent sets**; other subsets of S are called **dependent sets**. Note that the empty set is always independent.

Another example of matroids arise with numerical matrices: for any matrix M, let S be its set of columns, and I be the family of linearly independent subsets of columns. Call this the **matrix matroid** of M. The terminology of independence comes from this setting. This was the motivation of Whitney, who coined the term 'matroid'.

The explicit enumeration of the set I is usually out of the question. So, in computational problems whose input is a matroid (S, I), the matroid is usually implicitly represented. The above examples illustrate this: a graphic matroid is represented by a graph G, and the matrix matroid is represented by a matrix M. The size of the input is then taken to be the size of G or M, not of $|I|$ which can exponentially larger.

Submatroids. Given matroids $M = (S, I)$ and $M' = (S', I')$, we call M' a **submatroid** of M if $S' \subseteq S$ and $I' \subseteq I$. There are two general methods to obtain submatroids, starting from a non-empty subset $R \subseteq S$:
(i) Induced submatroids. The **R-induced submatroid of M** is

$$M|R \;:=\; (R, I \cap 2^R).$$

(ii) Contracted[2] submatroids. The **R-contracted submatroid of M** is

$$M \wedge R \;:=\; (R, I \wedge R)$$

where $I \wedge R := \{A \cap R : A \in I, S - R \subseteq A\}$. Thus, there is a bijective correspondence between the independent sets A' of $M \wedge R$ and those independent sets A of M which contain $S - R$. Indeed, $A' = A \cap R$. Of course, if $S - R$ is dependent, then $I \wedge R$ is empty.

We leave it to an exercise to show that $M|R$ and $M \wedge R$ are matroids. Special cases of induced and contracted submatroids arise when $R = S - \{e\}$ for some $e \in S$. In this case, we say that $M|R$ is obtained by **deleting** e and $M \wedge R$ is obtained by **contracting** e.

[2]Contracted submatroids are introduced here for completeness. They are not used in the subsequent development (but the exercises refer to them).

Bases. Let $M = (S, I)$ be a matroid. If $A \subseteq B$ and $B \in I$ then we call B an **extension** of A; if $A = B$, the extension is **improper** and otherwise it is **proper**. A **base** of M (alternatively: a **maximal independent set**) is an independent set with no proper extensions. If $A \cup \{e\}$ is independent and $e \notin A$, we call $A \cup \{e\}$ a **simple extension** of A and say that e **extends** A. If $R \subseteq S$, we may relativize these concepts to R: we may speak of "$A \subseteq R$ being a base of R", "e extends A in R", etc. This is the same as viewing A as a set of the induced submatroid $M|R$.

Ranks. We note a simple property: *all bases of a matroid have the same size.* If A, B are bases and $|A| > |B|$ then there is an $e \in A - B$ such that $B \cup \{e\}$ is a simple extension of B. This is a contradiction. Note that this property is true even if S has infinite cardinality. Thus we may define the **rank** of a matroid M to be the size of its bases. More generally, we may define the rank of any $R \subseteq S$ to be the size of the bases of R (this size is just the rank of $M|R$). The **rank function**

$$r_M : 2^S \to \mathbb{N}$$

simply assigns the rank of $R \subseteq S$ to $r_M(R)$.

Problems on Matroids. A **costed matroid** is given by $M = (S, I; C)$ where (S, I) is a matroid and $C : S \to \mathbb{R}$. is a cost[3] function. The cost of a set $A \subseteq S$ is just the sum $\sum_{x \in A} C(x)$. The **maximum independent set problem** (abbreviated, MIS) is this: given a costed matroid $(S, I; C)$, find an independent set $A \subseteq S$ with maximum cost. A closely related problem is the **maximum base problem** where, given $(S, I; C)$, we want to find a base $B \subseteq S$ of maximum cost. If the costs are non-negative, then it is easy to see the MIS problem and the maximum base problem are identical. The following algorithm solves the maximum base problem:

Greedy Algorithm for Maximum Base:
Input: matroid $M = (S, I; C)$ with cost function C.
Output: a base $A \in I$ with maximum cost.
1. Sort $S = \{x_1, \ldots, x_n\}$ by cost.
   Suppose $C(x_1) \geq C(x_2) \geq \cdots \geq C(x_n)$.
2. Initialize $A \leftarrow \emptyset$.
3. For $i = 1$ to n,
   put x_i into A provided this does not make A dependent.
4. Return A.

The steps in this abstract algorithm needs to be instantiated for particular representations of matroids. In particular, testing if a set A is independent is usually non-trivial (recall that matroids are usually given implicitly in terms of other combinatorial structures). We discuss this issue for graphic matroids below. It is interesting to note that the usual Gaussian algorithm for computing the rank of a matrix is an instance of this algorithm where the cost $C(x)$ of each element x is unit.

Let us see why the greedy algorithm is correct.

Lemma 4 (Correctness). *Suppose the elements of A are put into A in this order:*

$$z_1, z_2, \ldots, z_m,$$

where $m = |A|$. Let $A_i = \{z_1, z_2, \ldots, z_i\}$, $i = 1, \ldots, m$. Then:
1. A is a base.

[3]Recall our convention that costs may be negative. If the costs are non-negative, we call C a a "weight function".

2. If $x \in S$ extends A_i then $i < m$ and $C(x) \leq C(z_{i+1})$.
3. Let $B = \{u_1, \ldots, u_k\}$ be an independent set where $C(u_1) \geq C(u_2) \geq \cdots \geq C(u_k)$. Then $k \leq m$ and $C(u_i) \leq C(z_i)$ for all i.

Proof. 1. By way of contradiction, suppose $x \in S$ extends A. Then $x \notin A$ and we must have decided not to place x into the set A at some point in the algorithm. That is, for some $j \leq m$, $A_j \cup \{x\}$ is dependent. This contradicts the hereditary property because $A_j \cup \{x\}$ is a subset of the independent set $A \cup \{x\}$.
2. Suppose x extends A_i. By part 1, $i < m$. If $C(x) > C(z_{i+1})$ then for some $j \leq i$, we must have decided not to place x into A_j. This means $A_j \cup \{x\}$ is dependent, which contradicts the hereditary property since $A_j \cup \{x\} \subseteq A_i \cup \{x\}$ and $A_i \cup \{x\}$ is independent.
3. Since all bases are independent sets with the maximum cardinality, we have $k \leq m$. The result is clearly true for $k = 1$ and assume the result holds inductively for $k - 1$. So $C(u_j) \leq C(z_j)$ for $j \leq k - 1$. We only need to show $C(u_k) \leq C(z_k)$. Since $|B| > |A_{k-1}|$, the exchange property says that there is an $x \in B - A_{k-1}$ that extends A_{k-1}. By part 2, $C(z_k) \geq C(x)$. But $C(x) \geq C(u_k)$, since u_k is the lightest element in B by assumption. Thus $C(u_k) \leq C(z_k)$, as desired. **Q.E.D.**

From this lemma, it is not hard to see that an algorithm for the MIS problem is obtained by replacing the for-loop ("for $i = 1$ to n") in the above Greedy algorithm by "for $i = 1$ to m" where x_m is the last positive element in the list $(x_1, \ldots, x_m, \ldots, x_n)$.

Greedoids. While the matroid structure allows the Greedy Algorithm to work, it turns out that a more general abstract structure called **greedoids** is tailor-fitted to the greedy approach. To see what this structure looks like, consider the set system (S, F) where S is a non-empty finite set, and $F \subseteq 2^S$. In this context, each $A \in F$ is called a **feasible set**. We call (S, F) a **greedoid** if

Accessibility property If A is a non-empty feasible set, then there is some $e \in A$ such that $A \setminus \{e\}$ is feasible.

Exchange property: If A, B are feasible and $|A| < |B|$ then there is some $e \in B \setminus A$ such that $A \cup \{e\}$ is feasible.

Exercises

Exercise 5.1: Consider the graphic matroid in figure 4. Determine its rank function. ♢

Exercise 5.2: The text described a modification of the Greedy Maximum Base Algorithm so that it will solve the MIS problem. Verify its correctness. ♢

Exercise 5.3:
(a) Interpret the induced and contracted submatroids $M|R$ and $M \wedge R$ in the bigraph of figure 4, for various choices of the edge set R. When is $M|R = M \wedge R$?
(b) Show that $M|R$ and $M \wedge R$ are matroids in general. ♢

Exercise 5.4: Show that $r_M(A \cup B) + r_M(A \cap B) \leq r_M(A) + r_M(B)$. This is called the **submodularity property** of the rank function. It is the basis of further generalizations of matroid theory. ♢

Exercise 5.5: (Gavril) Consider the **activities selection problem** in which we are given a set

$$S = \{A_1, A_2, \ldots, A_n\}$$

of intervals. Each A_i is the half-open interval $A_i = [s_i, f_i)$ which represents an "activity" that starts at time s_i and finishes just before time f_i. A subset $F \subseteq S$ is called a **solution** and its **size** is the number of activities in F. We say F is **feasible** if for all $A, B \in F$, if $A \neq B$ then $A \cap B = \emptyset$. We say F is **optimal** if its size is maximum among all feasible solutions. E.g., if $S = \{[1,3), [0,2), [2,4)\}$ then $\{[1,3), [0,2)\}$ is not feasible, and $\{[0,2), [2,4)\}$ is an optimal solution.
(a) Prove that the following greedy algorithm is correct: sort the intervals in order of non-decreasing finish times. After renumbering the intervals, we may assume $f_1 \leq f_2 \leq \cdots \leq f_n$. Now we consider A_1, A_2, etc, in turn. Each A_i is accepted iff it does not conflict with the previously accepted intervals.
(b) What is the running time of this algorithm? Note: in deciding if an A_i is in conflict, it is enough to only look at the last accepted interval.
(c) Does the collection of feasible sets form a matroid? If yes, prove it. If no, give a counter example.
♢

Exercise 5.6:
(a) The greedy solution to the above activities selection problem uses the "finish time greedy criterion": the smallest remaining f_i is selected for consideration. We could conceive of other (apparently reasonable) greedy criteria:

1. Order the intervals A_i $(i = 1, \ldots, n)$ in non-decreasing order of their lengths $f_i - s_i$.
2. Order A_i in non-decreasing order of s_i.
3. Order A_i in non-decreasing "degree of conflict" where the degree of conflict of A_i is the number of j's $(j \neq i)$ such that A_i, A_j conflict.

For each of these ordering, either prove that the greedy method works or else produce a counter example. Note: the greedy method says that for each item in the sorted list, pick it iff this will not cause infeasibility among the picked items.
(b) Surely there is some symmetry between start and finish times. Find the "start time greedy criterion" analogous to the "finish time greedy criterion'. ♢

Exercise 5.7: Again consider the activities selection problem. The **length** of a feasible solution F is $\sum_{A \in F} |A|$ where $|A|$ denotes the length $f - s$ of the interval $A = [s, f)$. If F is infeasible, then we define its length to be 0. Now, define a feasible solution to be **optimal** if its length is maximum. Let $S_{i,j} = \{A_i, A_{i+1}, \ldots, A_j\}$ for $i \leq j$ and $F_{i,j}$ be an optimal solution for $S_{i,j}$.
(a) Show by a counter-example that the following "dynamic programming principle" fails:

$$F_{i,j} = \overline{\max}_{i \leq k \leq j-1} F_{i,k} \cup F_{k+1,j}$$

where $\overline{\max}\{F_1, F_2, \ldots, F_m\}$ returns the set F_ℓ whose length is maximum. (Recall that the length of F_ℓ is zero if it is not feasible.
(b) Give an $O(n \log n)$ algorithm for this problem. HINT: order the activities in the set S according to their finish times, say,

$$f_1 \leq f_2 \leq \cdots \leq f_n.$$

Consider the set of subproblems $S_i := S_{1,i}$ for $i = 1, \ldots, n$. Use an incremental algorithm (solve $S_1, S_2, \ldots, S_n$ in this order). ♢

Exercise 5.8: Give a divide-and-conquer algorithm for the problem in previous exercise, to find the maximum length feasible solution for a set S of activities. (This approach is harder and less efficient!)
♢

Exercise 5.9: A vertex cover for a bigraph $G = (V, E)$ is a subset $C \subseteq V$ such that for all edge e in E, at least one of its two vertices is contained in C. A **minimum vertex cover** is one of minimum size. Here is a greedy algorithm that finds a vertex cover VC:

1. Initialize VC to the empty set and initialize G' to the input graph.
2. While the edge set of G' is not empty: Select a vertex v of maximum degree, add v to the set VC, and remove v and all edges incident on v from G'.
3. Output VC.

Show that this greedy algorithm may fail to find a minimum vertex cover. EXTRA CREDIT: It is OK to give an example in which the greedy algorithm *may* find a suboptimal solution, depending on how it breaks ties when two or more vertices have the same degree. But you get extra credit if the algorithm is *guaranteed* to find a suboptimal solution on your example. An example with 7 vertices exists. ♢

End Exercises

§6. Minimum Spanning Tree

The Minimum Base Problem. Consider the **minimum base problem** for a costed matroid $(S, I; C)$ where C is a cost function $C : S \to \mathbb{R}$. The cost of a set $B \subseteq S$ is given by $\sum_{x \in B} C(x)$. So we want to compute a base $B \in I$ of minimum cost. A greedy algorithm is easily derived from the previous Greedy Algorithm for Maximum Base: we only have to replace the for-loop ("for $i = 1$ to n") by "for $i = n$ downto 1". We leave the justification for an exercise.

The **minimum spanning forest problem** is an instance of the minimum base problem. Here we are given a costed bigraph

$$G = (V, E; C)$$

where $C : E \to \mathbb{R}$. In the previous section, we show that the set I of acyclic sets of G is a matroid. An acyclic set $T \subseteq E$ of maximum cardinality is called a **spanning forest**; in this case, $|T| = |V| - c$ where G has $c \geq 1$ components. The **cost** $C(T)$ of any subset $T \subseteq E$ is given by $C(T) = \sum_{e \in T} C(e)$. An acyclic set is **minimum** if its cost is minimum. It is conventional to make the following simplification:

The input bigraph G is connected.

In this case, a spanning forest T is actually a tree, and the problem is known as the **minimum spanning tree (MST) problem**. The simplification is not too severe: if our graph is not connected, we can first compute its connected component (another basic graph problem that has efficient solution) and then apply the MST algorithm to each component. Alternatively, it is not hard to modify an MST algorithm so that it applies even if the input is not connected.

Consider the bigraph in figure 4 with vertices $V = \{a, b, c, d, e\}$. One such MST is $\{b{-}c, d{-}e, a{-}c, a{-}e\}$, with cost 6. It is easy to verify that there are six MST's, as shown in figure 5.

The greedy method for minimum bases is applicable to the MST problem. The minimum base algorithm, restated for MST, is called **Kruskal's algorithm**. Here is the description: order the m edges of the input G so that

$$C(e_1) \leq C(e_2) \leq \cdots \leq C(e_m) \tag{7}$$

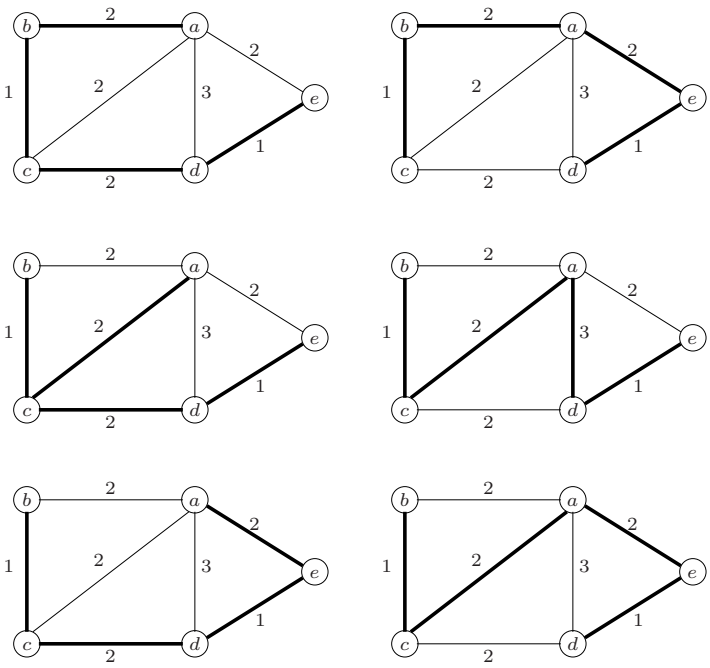


Figure 5: MST's of a bigraph.

and for each $i = 1, \ldots, m$ in turn, we **accept** e_i provided it does not create a cycle with the previously accepted edges.

Actually, Kruskal's algorithm is an instance of a general schema for the greedy MST algorithms:

Generic Greedy MST Algorithm
Input: $G = (V, E; C)$ a connected bigraph with edge costs.
Output: $S \subseteq E$, a MST for G.
$S \leftarrow \emptyset$.
for $i = 1$ to $|V| - 1$ do
1. Find an $e \in E - S$ that is "safe for S".
2. $S \leftarrow S + e$.
Output S as the minimum spanning tree.

NOTATION: it is convenient to write "$S + e$" for "$S \cup \{e\}$" in this discussion. Likewise, "$S - e$" shall denote the set "$S \setminus \{e\}$".

What does it mean for "e to be safe for S"? Surely, it is sufficient if $S + e$ is contained in some MST. But this criteria seems hard to characterize in a computationally effective way. Various instances of the above generic algorithm amount to defining some other criterion which is computationally effective.

Let us say that e is a **candidate** for S if $S + e$ is acyclic. If U is a connected component of $G' = (V, S)$, and $e = (u, v)$ is a candidate such that $u \in U$ or $v \in U$ then we say that e **extends** U. Note that if e extends U then the graph $G'' = (V, S + e)$ will not have U as a component.

The following are 4 notions of what it means for "e to be safe for S":

- (Simple) $S + e$ is extendible to some MST. This, as we said, is computational ineffective.
- (Kruskal) Edge e has the least cost among all the candidates.
- (Boruvka) There is a component U of $G' = (V, S)$ such that e has the least cost among all the candidates that extend U.
- (Prim) This has, in addition to Boruvka's condition, the requirement that the graph $G'' = (V, S + e)$ has only one non-trivial component. [A component is trivial if it has only a single vertex.]

Let us call those sets $S \subseteq E$ that may arise during the execution of the generic MST algorithm **simply-safe**, **Boruvka-safe**, **Kruskal-safe** or **Prim-safe**, depending on which of the above definition of safety is used.

The latter three criteria are named for the inventors of three versions of the generic MST algorithm. The correctness of these algorithms amounts to showing that "*X-safe implies simply-safe*" where X = Kruskal, Boruvka or Prim. The previous section has essentially shown the correctness of Kruskals's algorithm. Let us now show the correctness of the algorithm of Boruvka. By definition, Prim-safe implies Boruvka-safe, and so Prim's algorithm is also correct. Indeed, Kruskal-safe also implies Boruvka-safely, so we obtain an alternative proof of correctness for Kruskal's algorithm.

Lemma 5 (Correctness of Boruvka's Algorithm). *Boruvka-safe sets are simply-safe.*

Proof. We use induction on the size $|S|$ of Boruvka-safe sets S. Clearly if $S = \emptyset$, then S is Boruvka-safe and this is clearly simply-safe. Next suppose $S = S' + e$ where S' is Boruvka-safe. We need to prove that S is simply-safe. By definition of Boruvka-safety, there is a component U of the graph $G' = (V, S')$ such that e has the least cost among all edges that extend U. By induction hypothesis, we may assume S' is simply-safe. Hence there is a MST T' that contains S'. If $e \in T'$, then we are done (as T' would be a witness to the fact that $S = S' + e$ is simply-safe). So assume $e \notin T'$.

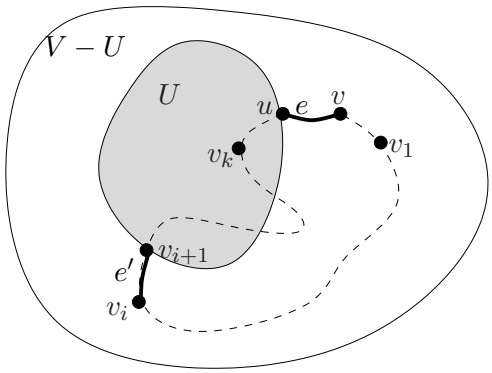


Figure 6: Extending a component U by $e = (u, v)$.

Write $e = (u, v)$ such that $u \in U$ and $v \notin U$. Hence $T' + e$ contains a unique closed path of the form

$$Z := (u{-}v{-}v_1{-}v_2{-}\cdots{-}v_k{-}u).$$

There exists some $i = 0, \ldots, k$ such that $v_i \notin U$ and $v_{i+1} \in U$. Write

$$Z = (u{-}v{-}v_1{-}\cdots{-}v_i{-}v_{i+1}{-}\cdots{-}u)$$

(where $v = v_0$ and $u = v_{k+1}$ in this notation). Let $e' := (v_i{-}v_{i+1})$. Note that $T := T' + e - e'$ is acyclic and is a spanning tree. Moreover, $C(e) \leq C(e')$, by our choice of e. Hence $C(T) \leq C(T')$ and so T is a MST. This shows that S is simply-safe, as S contains T. **Q.E.D.**

Next, we need effective implementations of the above notions of safety. In the case of Prim's algorithm, this is taken up in Chapter 6 (amortization techniques). Similarly, we will show how to implement Kruskal's algorithm in Chapter 12 when we study the union-find data structure. However, an exercise below will lead you through a method of organizing a hand-simulation of Prim's and Kruskal's algorithm so that you understand the mechanics involved in these two algorithms.

Safe sets of vertices. Let us define the notion of "safety" for sets of vertices. For any set $S \subseteq E$ of edges, let $V(S)$ denote the set of those vertices that are incident on some edge of S. We say a set $U \subseteq V$ is X**-safe** if there exists an X-safe set $S \subseteq E$ such that $U = V(S)$. Here, X is equal to 'simply', 'Prim', 'Kruskal' or 'Boruvka'. By this definition, no singleton would be safe. Instead, we define safety for singletons thus: a singleton $\{v\}$ is defined to be X**-safe** if there exists u such that $\{u, v\}$ is X-safe by the previous definition.

Hand Simulation of MST Algorithms. We expect students to do hand simulation of Kruskal's and Prim's algorithms. The trick is devise a compact way to represent the intermediate steps. For Kruskal's algorithm, this is easy – we just list the edges by non-decreasing weight order and indicate the acceptance/rejection of successive edges.

For Prim's algorithm, we just maintain an array $d[1..n]$ assuming the vertex set is $V = \{1, \ldots, n\}$. We shall maintain a subset $S \subseteq V$ representing the set of vertices which we know how to connect to the source node 1 in a MST. The set S is "Prim safe". Initially, let $S = \emptyset$ and $d[1] = 0$ and $d[v] = \infty$ for $v = 2, \ldots, n$. In general, the entry $d[v]$ ($v \in V \setminus S$) represents the "cheapest" cost to connect vertex v to the MST on the set S. Our simulation consists in building up a matrix M which is a $n \times n$ matrix, where the 0th row representing the initial array d. Each time the array d is updated, we rewrite it as a new row of a matrix M.

At stage $i \geq 1$, suppose we pick a node $v_i \in V \setminus S$ where $d[v_i] = \min\{d[j] : j \in V \setminus S\}$. We add v_i to S, and update all the values $d[u]$ for each $u \in V \setminus S$ that is adjacent to v_i. The update rule is this:

$$d[u] = \min\{d[u], COST[v_i, u]\}.$$

The resulting array is written as row i in our matrix.

Let us illustrate the process on the graph of Figure 7. The vertex set is $V = \{1, 2, \ldots, 11, 12\}$. The final matrix is the following:

Stage	1	2	3	4	5	6	7	8	9	10	11	12
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	X	3	1	7	∞	∞	∞	∞	∞	∞	∞	∞
2			X	6				3				
3		X			6							
4								X	8			
5				X		7						
6					X		6					6
7						3	X			3	2	
8						1				1	X	2
9						X						
10									6	X		
11												X
12									X			

Some conventions: We mark the newly picked node in each stage with an 'X'. Also, any value that is

unchanged from the previous row may be left blank. Thus, in stage 2, the node 3 is picked and we update $d[4]$ using $d[4] = \min\{d[4], COST[3,4]\} = \min\{7, 6\} = 6$.

The final cost of the MST is 37. To see this, each X corresponds to a vertex v that was picked, and the last value of $d[v]$ contributes to the cost of the MST. E.g., the X corresponding to vertex 1 has cost 0, the X corresponding to vertex 2 has cost 3, etc. Summing up over all X's, we get 37.

Remarks: Boruvka (1926) has the first MST algorithm. The algorithm attributed to Prim (1957) was discovered earlier by Jarník (1930). These algorithms have been rediscovered many times. See [5] for further references. Both Boruvka and Jarník's work are in Czech. The Prim-Jarník algorithm is very similar in structure to Dijkstra's algorithm which we will encounter in the chapter on minimum cost paths.

Exercises

Exercise 6.1: We consider minimum spanning trees (MST's) in an undirected graph $G = (V, E)$ where each vertex $v \in V$ is given a numerical value $C(v) \geq 0$. The **cost** $C(u, v)$ of an edge $(u{-}v) \in E$ is defined to be $C(u) + C(v)$.
(a) Let G be the graph in figure 7. The value $C(v)$ is written next to the node v. For instance

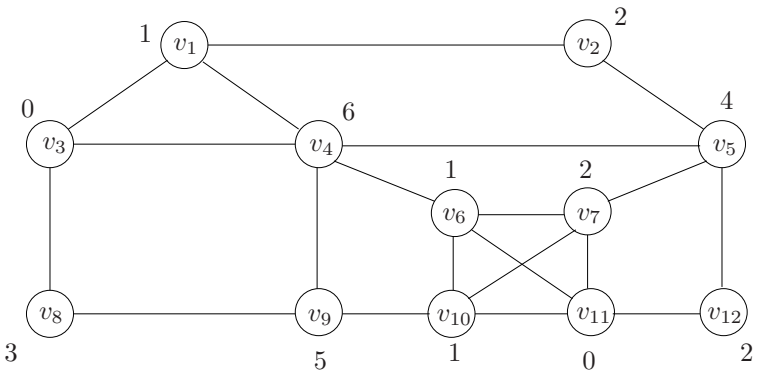


Figure 7: The house graph.

$C(v_4) = 6$ and $C(v_1, v_4) = 1 + 6 = 7$. Compute an MST of G using Boruvka's algorithm. Please organize your computation so that we can verify intermediate results. Also state the cost of your minimum spanning tree.
(b) Can you design a special algorithm for MST in which edge costs has the special form $C(u, v) = C(u) + C(v)$ as in part(a)? ♢

Exercise 6.2: Suppose G is the complete bipartite graph $G_{m,n}$. That is, the vertices V are partitioned into two subsets V_0 and V_1 where $|V_0| = m$ and $V_1| = n$ and $E = V_0 \times V_1$. Give a simple description of an MST of $G_{m,n}$. Argue that your description is indeed an MST. HINT: transform an arbitrary MST into your description by modifying one edge at a time. ♢

Exercise 6.3: Let G_n be the bigraph whose vertices are $V = \{1, 2, \ldots, n\}$. The edges are defined as follows: for each $i \in V$, if i is prime, then $(1, i) \in E$ with weight i. [Recall that 1 is not considered prime, so 2 is the smallest prime.] For $1 < i < j$, if i divides j then we add (i, j) to E with weight j/i.
(a) Draw the graph G_{19}.

(b) Compute the MST of G_{18} using Prim's algorithm, using node 1 as the source vertex. Please use the organization described in the appendix below. ♢

Exercise 6.4: Describe the rule for reconstructing the MST from the matrix M using in our hand-simulation of Prim's Algorithm. ♢

Exercise 6.5: Hand Simulation of Kruskal's Algorithm on the graph of Figure 7. This exercise suggests a method for carry out the steps of this algorithm. The edges in sorted order are shown in the table below.

Next, we now consider each edge in turn. We maintain a partition of $V = \{1, \ldots, 12\}$ into disjoint sets. Let $L(i)$ denote the set containing vertex i. Initially, each node is in its own set, i.e., $L(i) = \{i\}$. Whenever an edge i–j is added to the MST, we merge the corresponding sets $L(i) \cup L(j)$. E.g., in the first step, we add edge 1–3. Thus the lists $L(1) = \{1\}$ and $L(3) = \{1\}$ are merged, and we get $L(1) = L(3) = \{1, 3\}$. To show the computation of Kruskal's algorithm, for each edge, if the edge is "rejected", we mark it with an "X". Otherwise, we indicate the merged list resulting from the union of $L(i)$ and $L(j)$: Please fill in the last two columns of the table (we have filled in the first 4 rows for you).

Sorting Order	Edge	Weight	Merged List	Cumulative Weight
1	1–3:	1	$\{1, 3\}$	1
2	6–11:	1	$\{6, 11\}$	2
3	10–11:	1	$\{6, 10, 11\}$	3
4	6–10:	2	X	3
5	7–11:	2		
6	11–12:	2		
7	1–2:	3		
8	3–8:	3		
9	6–7:	3		
10	7–10:	3		
11	2–5:	6		
12	3–4:	6		
13	5–7:	6		
14	5–12:	6		
15	9–10:	6		
16	1–4:	7		
17	4–6:	7		
18	8–9:	8		
19	4–5:	10		
20	4–9:	11		

♢

Exercise 6.6: This question considers two concrete ways to implement Kruskal's algorithm. Let $V = \{1, 2, \ldots, n\}$ and $D[1..n]$ be an array of size n that represents a **forest** $G(D)$ with vertex set V and edge set $E = \{(i, D[i]) : i \in V\}$. More precisely, $G(D)$ is an directed graph that has no cycles except for self-loops (i.e., edges of the form (i, i)). A vertex i such that $D[i] = i$ is called a **root**. The set V is thereby partitioned into disjoint subsets $V = V_1 \cup V_2 \cup \cdots \cup V_k$ (for some $k \geq 1$) such that each V_i has a unique root r_i, and from every $j \in V_i$ there is a path from j to r_i. For example, with $n = 7$, $D[1] = D[2] = D[3] = 3$, $D[4] = 4$, $D[5] = D[6] = 5$ and $D[7] = 6$ (see Figure 8). We call V_i a **component** of the graph $G(D)$ (this terminology is justified because V_i is a component in the usual sense if we view $G(D)$ as an *undirected* graph).

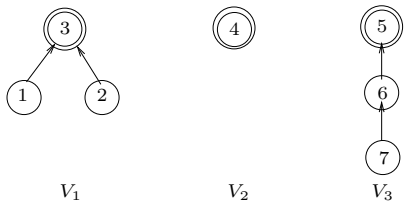


Figure 8: Directed graph $G(D)$ with three components (V_1, V_2, V_3)

(i) Consider two restrictions on our data structure: Say D is **list type** if each component is a linear list. Say D is **star type** if each component is a star (i.e., each vertex in the component points to the root). E.g., in Figure 8, V_2 and V_3 are linear lists, while V_1 and V_2 are stars. Let ROOT(i) denote the root r of the component containing i. Give a pseudo-code for computing ROOT(i), and give its complexity in the 2 cases: (1) D is list type, (2) D is star type.
(ii) Let COMP(i) $\subseteq V$ denote the component that contains i. Define the operation $MERGE(i, j)$ that transforms D so that COMP(i) and COMP(j) are combined into a new component (but all the other components are unchanged). E.g., the components in Figure 8 are $\{1, 2, 3\}, \{4\}$ and $\{5, 6, 7\}$. After $MERGE(1, 4)$, we have two componets, $\{1, 2, 3, 4\}$ and $\{5, 6, 7\}$. Give a pseudo-code that implements $MERGE(i, j)$ under the assumption that i, j are roots and D is list type which you must preserve. Your algorithm *must* have complexity $O(1)$. To achieve this complexity, you need to maintain some additional information (perhaps by a simple modification of D).
(iii) Similarly to part (ii), implement $MERGE(i, j)$ when D is star type. Give the complexity of your algorithm.
(iv) Describe how to use ROOT(i) and $MERGE(i, j)$ to implement Kruskal's algorithm for computing the minimum spanning tree (MST) of a weighted connected undirected graph H.
(v) What is the complexity of Kruskal's in part (iv) if (1) D is list type, and if (2) D is star type. Assume H has n vertices and m edges. ♢

Exercise 6.7: Give two alternative proofs that the suggested algorithm for computing minimum base is correct:
(a) By verifying the analogue of the Correctness Lemma.
(b) By replacing the cost $C(e)$ (for each $e \in E$) by the cost $c_0 - C(e)$. Choose c_0 large enough so that $c_0 - C(e) > 0$. ♢

Exercise 6.8: Let G be a bigraph G with distinct weights.
(a) Prove that the minimal spanning tree T of an must contain that edge of smallest weight.
(b) Must it contain the edge of second smallest weight?
(c) Must it contain the edge of third smallest weight?
(d) Student Quick observed that Kruskal's algorithm does pick the edges of smallest and second smallest weights. Since Kruskal's algorithm correctly computes the MST this proves (a) and (b). What is missing in this argument? ♢

Exercise 6.9: Show that every MST can be obtained from Kruskal's algorithm by a suitable re-ordering of the edges which have identical weights. Conclude that when the edge weights are unique, then the MST is unique. ♢

Exercise 6.10: Student Joe wants to reduce the minimum base problem for a costed matroid $(S, I; C)$ to the MIS problem for $(S, I; C')$ where C' is a suitable transformation of C.
(a) Student Joe considers the modified cost function $C'(e) = 1/C(e)$ for each e. Construct an example

to show that the MIS solution for C' need not be the same as the minimum base solution for C.
(b) Next, student Joe considers another variation: he now defines $C'(e) = -C(e)$ for each e. Again, provide a counter example. ♢

Exercise 6.11: Extend the algorithm to finding MIS in contracted matroids. ♢

Exercise 6.12: If $S \subseteq E$ is Prim-safe, then clearly $G' = (V(S), S)$ is clearly a tree. Prove that S is actually an MST of the restricted graph $G|V(S)$. ♢

Exercise 6.13:
(a) Enumerate the X-safe sets of vertices in figure 4. Here, X is 'simply', 'Kruskal', 'Boruvka' or 'Prim'.
(b) Characterize the safe singletons (relative to any of the three notions of safety). ♢

Exercise 6.14: (Tarjan) Consider the following **generic accept/reject algorithm** for MST. This consists of steps that either **accept** or **reject** edges. In our generic MST algorithm, we only explicitly accept edges. However, we may be implicitly rejecting edges as well, as in the case of Kruskal's algorithm. Let S, R be the sets of accepted and rejected edges (so far). We say that (S, R) is **simply-safe** if there is an MST that contains S but not containing any edge of R. Note that this extends our original definition of "simply safe". Prove that the following extensions of S and R will maintain minimal safety:
(a) Let $U \subseteq V$ be any subset of vertices. The set of edges of the form (u, v) where $u \in U$ and $v \notin U$ is called a U-**cut**. If e is the minimum cost edge of a U-cut and there are no accepted edges in the U-cut, then we may extend S by e.
(b) If e is the maximum cost edge in a cycle C and there are no rejected edges in C then we may extend R by e. ♢

Exercise 6.15: With respect to the generic accept/reject version of MST:
(a) Give a counter example to the following rejection rule: let e and e' be two edges in a U-cut. If $C(e) \geq C(e')$ then we may reject e'.
(b) Can the rule in part (a) be fixed by some additional properties that we can maintain?
(c) Can you make the criterion for rejection in the previous exercise (part (b)) computationally effective? Try to invent the "inverses" of Prim's and Boruvka's algorithm in which we solely reject edges.
(d) Is it always a bad idea to *only* reject edges? Suppose that we alternatively accept and reject edges. Is there some situation where this can be a win? ♢

Exercise 6.16: Consider the following recursive "MST algorithm" on input $G = (V, E; C)$:
(I) Subdivide $V = V_1 \uplus V_2$.
(II) Recursive find a "MST" T_i of $G|V_i$ $(i = 1, 2)$.
(III) Find e in the V_1-cut of minimum cost. Return $T_1 + e + T_2$.
Give a small counter-example to this algorithm. Can you fix this algorithm? ♢

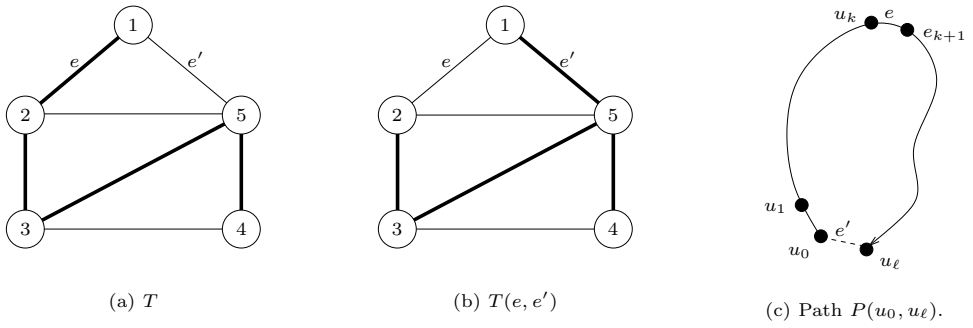
Exercise 6.17: Is there an analogue of Prim and Boruvka's algorithm for the MIS problem for matroids? ♢

Exercise 6.18: Let $G = (V, E; C)$ be the complete graph in which each vertex $v \in V$ is a point in the Euclidean plane and $C(u, v)$ is just the Euclidean distance between the points u and v. Give efficient methods to compute the MST for G. ♢

Exercise 6.19: Joe Moe thought that a simple way to compute the MST is to pick, for each vertex v, the edge $(v{-}u)$ that has the least cost among all the nodes u that are adjacent to v. Let P be the set of edges so picked.
(a) Show that $n/2 \leq P \leq n-1$. Give examples where these two extreme bounds are achieved (your examples must be described in general terms for every n).
(b) Show that if the costs are unique, P cannot contain a cycle. What kinds of cycles can form if weights are not unique?
(c) Assume vertices in P is picked with the tie breaking rule: when two or more vertices can be picked, choose the smallest numbered vertex (assume vertices are numbered from 1 to n). This clearly avoids cycles. Prove that P has the following property: if add an edge e to P creates a cycle Z in P, then e has the maximum cost among the edges in Z.
(d) For any costed bigraph $G = (V, E; C)$, and $P \subseteq E$, we define a new costed bigraph denoted G/P. First, two vertices of V are said to be equivalent modulo P if they are connected by a sequence of edges in P. For $v \in V$, let $[v]$ denote the equivalence class of v. The vertices of G/P is the set $\{[v] : vinV\}$. The edges of G/P are those $([u]{-}[v])$ such that there exists some $u' \in [u]$ and $v' \in [v]$ with $(u'{-}v') \in E$. The cost of $([u]{-}[v])$ is the minimum cost in the set $\{C(u', v') : u' \in [u], v' \in [v], (u'{-}v') \in E\}$. Note that G/P has at most $n/2$ vertices. Moreover, we can pick another set P' of edges in G/P using the same rules as before. This gives us another graph $(G/P)/P'$ with at most $n/4$ vertices. We can continue this until V has 1 vertex. Briefly describe how this gives us another MST algorithm. You must show how to recover the MST in your algorithm. What is the complexity of your algorithm?

♢

Exercise 6.20: Fix a connected undirected graph $G = (V, E)$. Let $T \subseteq E$ be any spanning tree of G. A pair (e, e') of edges is called a **swappable pair for** T if
(i) $e \in T$ and $e' \in E \setminus T$ (Notation: for sets A, B, their difference is denoted $A \setminus B = \{a \in A : a \notin B\}$)
(ii) The set $(T \setminus \{e\}) \cup \{e'\}$ is a spanning tree.
Let $T(e, e')$ denote the spanning tree $(T \setminus \{e\}) \cup \{e'\}$ obtained from T **by swapping** e **and** e' (see illustration in Figure 9(a), (b)).



(a) T (b) $T(e, e')$ (c) Path $P(u_0, u_\ell)$.

Figure 9: (a) A swappable pair (e, e') for spanning tree T. (b) The new spanning tree $T(e, e')$ [NOTE: tree edges are indicated by thick lines]

(a) Suppose (e, e') is a swappable pair for T and $e' = (u, v)$. Prove that e lies on the unique path, denoted by $P(u, v)$, of T from u to v. In Figure 9(a), $e' = (1{-}5) = (5{-}1)$. So the path is either $P(1, 5) = (1{-}2{-}3{-}5)$ or $P(5, 1) = (5{-}3{-}2{-}1)$.
(b) Let $n = |V|$. Relative to T, we define a $n \times n$ matrix $First$ indexed by pairs of vertices u, v, where $First[u, v] = w$ means that the first edge in the unique path $P(u, v)$ is (u, w). (In the special case of $u = v$, let $First[u, u] = u$.) In Figure 9(a), $First[1, 5] = 2$ and $First[5, 1] = 3$. Show the matrix $First$ for the tree T in Figure 9(a). Similarly, give the matrix $First$ for the tree $T(e, e')$ in Figure 9(b).
(c) Describe an $O(n^2)$ algorithm called $Update(First, e, e')$ which updates the matrix $First$ after we transform T to $T(e, e')$. HINT: For which pair of vertices (x, y) does the value of $First[x, y]$ have to

change? Suppose $e' = (u', v')$ and $P(u', v') = (u_0, u_1, \ldots, u_\ell)$ is as illustrated in Figure 9(c). Then $u' = u_0, v' = u_\ell$, and also $e = (u_k, u_{k+1})$ for some $0 \leq k < \ell$. Then, originally $First[u_0, u_\ell] = u_1$ but after the swap, $First[u_0, u_\ell] = u_\ell$. What else must change?
(d) Analyze your algorithm to show that that it is $O(n^2)$. Be sure that your description in (c) is clear enough to support this analysis. ♢

End Exercises

§7. Generating Permutations

In §1, we saw how the general bin packing problem can be reduced to linear bin packing. This reduction depends on the ability to generate all permutations of n elements, in $O(n!)$ time. There are many other applications of such a permutation generator, so we now take a small detour to address this interesting topic. A survey of this classic problem is given by Sedgewick [4]. Perhaps the oldest incarnation of this problem is the "change ringing problem" of bell-ringers in early 17th Century English churches [3]. This calls for ringing a sequence of n bells in all $n!$ permutations.

The problem of generating all permutations efficiently is representative of an important class of problems called **combinatorial enumeration**. For instance, we might want to general all size k subsets of a set, all graphs of size n, all convex polytopes with n vertices, etc. Such an enumerations would be considered optimal if the algorithm takes $O(1)$ time to generate each member.

It is good to fix some terminology. A **n-permutation** of a finite set X is a surjective function $p : \{1, \ldots, n\} \to X$. Surjectivity of p implies $n \geq |X|$. The function p may be represented by a sequence $(p(1), p(2), \ldots, p(n))$. Here we are interested in the case $n = |X|$, *i.e.*, permutation of *distinct* elements. We use a path-like notation for permutations, writing "$(p(1)-\cdots-p(n))$" for the permutation $(p(1), p(2), \ldots, p(n))$.

Let S_n denote the set of all permutations of $X = \{1, 2, \ldots, n\}$; each element of S_n is called an **n-permutation**. Note that $|S_n| = n!$. E.g., the following is a listing of S_3:

$$(1{-}2{-}3), \quad (1{-}3{-}2), \quad (3{-}1{-}2); \quad (3{-}2{-}1), \quad (2{-}3{-}1), \quad (2{-}1{-}3). \tag{8}$$

Two n-permutations $\pi = (x_1-\cdots-x_n)$ and $\pi' = (x'_1-\cdots-x'_n)$ are said to be **adjacent** (to each other) if there is some $i = 2, \ldots, n$ such that $x_{i-1} = x'_i$ and $x_i = x'_{i-1}$, and for all other j, $x_j = x'_j$. Indeed, we write $\pi' = Exch_i(\pi)$ in this case. E.g., $\pi = (1{-}2{-}4{-}3)$ and $\pi' = (1{-}4{-}2{-}3)$ are adjacent since $\pi' = Exch_3(\pi)$. An **adjacency ordering** of a set S of permutations is a listing of the elements of S such that every two consecutive permutations in this listing are adjacent. For instance, the listing of S_3 in (8) is an adjacency ordering.

[Figure: Adjacency Graph for 3-permutations]

We need another concept: if $\pi = (x_1-\cdots-x_{n-1})$ is an $(n-1)$-permutation, and π' is obtained from π by inserting the letter n into π, then we call π' an **extension** of π. Indeed, if n is inserted just before the ith letter in π, then we write $\pi' = Ext_i(\pi)$ for $i = 1, \ldots, n$. The meaning of "$Ext_n(\pi)$" should be clear: it is obtained by appending 'n' to the end of the sequence π. Note that there are n extensions of π. E.g., if $\pi = (1{-}2)$ then the three extensions of π are $(3{-}1{-}2), (1{-}3{-}2), (1{-}2{-}3)$.

The Johnson-Trotter Ordering. Among the several known methods to generate all n-permutations, we will describe one that is independently discovered by S.M.Johnson and H.F.Trotter (1962), and apparently

known to 17th Century English bell-ringers [3]. The two main ideas in the Johnson-Trotter algorithm are (1) the n-permutations are generated as an adjacency ordering, and (2) the n-permutations are generated recursively. Suppose let π is an $(n-1)$-permutation that has been recursively generated. Then we note that the n extensions of π can given one of two adjacency orderings. It is either

$$UP(\pi) : Ext_1(\pi), Ext_2(\pi), \ldots, Ext_n(\pi)$$

or the reverse sequence

$$DOWN(\pi) : Ext_n(\pi), Ext_{n-1}(\pi), \ldots, Ext_1(\pi).$$

E.g., $UP(1{-}2{-}3)$ is equal to

$$(4{-}1{-}2{-}3), \quad (1{-}4{-}2{-}3), \quad (1{-}2{-}4{-}3), \quad (1{-}2{-}3{-}4).$$

Note that if π' is another $(n-1)$-permutation that is adjacent to π, then the concatenated sequences

$$UP(\pi); DOWN(\pi')$$

and

$$DOWN(\pi); UP(\pi')$$

are both adjacency orderings. We have thus shown:

Lemma 6 (Johnson-Trotter ordering). *If $\pi_1, \ldots, \pi_{(n-1)!}$ is an adjacency ordering of S_{n-1}, then the concatenation of alternating DOWN/UP sequences*

$$DOWN(\pi_1); UP(\pi_2); DOWN(\pi_3); \cdots; DOWN(\pi_{(n-1)!})$$

is an adjacency ordering of S_n.

For example, starting from the adjacency ordering of 2-permutations ($\pi_1 = (1{-}2), \pi_2 = (2{-}1)$), our above lemma says that $DOWN(\pi_1), UP(\pi_2)$ is an adjacency ordering. Indeed, this is the ordering shown in (8).

Let us define the **permutation graph** G_n to be the bigraph whose vertex set is S_n and whose edges comprise those pairs of vertices that are adjacent in the sense defined for permutations. We note that the adjacency ordering produced by Lemma 6 is actually a cycle in the graph G_n. In other words, the adjacency ordering has the additional property that the first and the last permutations of the ordering are themselves adjacent. A cycle that goes through every vertex of a graph is said to be **Hamiltonian**. If $(\pi_1{-}\pi_2{-}\cdots{-}\pi_m)$ (for $m = (n-1)!$) is a Hamiltonion cycle for G_{n-1}, then it is easy to see that

$$(DOWN(\pi_1); UP(\pi_2); \cdots; UP(\pi_m))$$

is a Hamiltonian cycle for G_n.

The Permutation Generator. We proceed to derive an efficient means to generate successive permutations in the Johnson-Trotter ordering. We need an appropriate high level view of this generator. The generated permutations are to be used by some "permutation consumer" such as our greedy linear bin packing algorithm. There are two alternative views of the relation between the "permutation generator" and the "permutation consumer". We may view the consumer as calling[4] the generator repeatedly, where each call to the generator returns the next permutation. Alternatively, we view the generator as a skeleton program with the consumer program as a (shell) subroutine. We prefer the latter view, since this fits the established paradigm of BFS and DFS as skeleton programs (see Chapter 4). Indeed, we may view the permutation generator as a bigraph traversal: the implicit bigraph here is the permutation graph G_n.

[4]The generator in this viewpoint is a **co-routine**. It has to remember its state from the previous call.

In the following, an n-permutation is represented by the array $per[1..n]$. We will transform per by exchange of two adjacent values, indicated by

$$per[i] \Leftrightarrow per[i-1] \tag{9}$$

for some $i = 2, \ldots, n$, or

$$per[i] \Leftrightarrow per[i+1]$$

where $i = 1, \ldots, n-1$.

A Counter for n factorial. To keep track of the successive exchanges in Johnson-Trotter generator, we introduce an array of n counters

$$C[1..n]$$

where each $C[i]$ is initiallized to 1 but always satisfying the relation $1 \leq C[i] \leq i$. Of course, $C[1]$ may be omitted since its value cannot change under our restrictions. The array counter C has $n!$ distinct possible. We say the i-th counter is **full** iff $C[i] = i$. The **level** of the C is the largest index ℓ such that the ℓ-th counter is not full. If all the counters are full, the level of C is defined to be 1. E.g., $C[1..5] = [1, 2, 2, 1, 5]$ has level 4. We define the **increment** of this counter array as follows: if the level of the counter is ℓ, then (1) we increment $C[\ell]$ provided $\ell > 1$, and (2) we set $C[i] = 1$ for all $i > \ell$. E.g., the increment of $C[1..5] = [1, 2, 2, 1, 5]$ gives $[1, 2, 2, 2, 1]$. In code:

```
Inc(C)
    ℓ ← n.
    while (C[ℓ] = ℓ) ∧ (ℓ > 1)
        C[ℓ--] ← 1.
    if (ℓ > 1)
        C[ℓ]++.
    return(ℓ)
```

Note that Inc returns the level of the original counter value. This subroutine is a generalization of the usual incrementation of binary counters (Chapter 6.1). For instance, for $n = 4$, starting with the initial value of $[1, 1, 1]$, successive increments of this array produce the following cyclic sequence:

$$\begin{aligned} C[2,3,4] &= [1,1,1] \to [1,1,2] \to [1,1,3] \to [1,1,4] \to [1,2,1] \\ &\to [1,2,2] \to [1,2,3] \to [1,2,4] \to [1,3,1] \to \cdots \\ &\to [2,3,3] \to [2,3,4] \to [1,1,1] \to \cdots . \end{aligned} \tag{10}$$

Let the cost of incrementing the counter array be equal to $n+1-\ell$ where ℓ is the level. CLAIM: the cost to increment the counter array from $[1, 1, \ldots, 1]$ to $[2, 3, \ldots, n]$ is $< 2(n!)$. In proof, note that $C[\ell]$ is updated after every $n!/\ell!$ steps, so that the overall, $C[\ell]$ is updated $\ell!$ times. Hence the total number of updates for the $n-1$ counters is

$$n! + (n-1)! + \cdots + 2! < 2(n!),$$

which proves our Claim.

This gives us the top level structure for our permutation generator:

```
Johnson-Trotter Generator (Sketch)
Input: natural number n ≥ 2
    ▷ Initialization
        per[1..n] ← [1, 2, ..., n].    ◁ Initial permutation
        C[2..n] ← [1, 1, ..., 1].    ◁ Initial counter value
    ▷ Main Loop
        do
            ℓ ← Inc(C)
            UPDATE(ℓ)    ◁ The permutation is updates
            CONSUME(per)    ◁ Permutation is consumed
        while (ℓ > 1)
```

The shell routine CONSUME is application-dependent. In illustrations, we simply use it to print the current permutation.

How to update the permutation. We now describe the UPDATE routine. It uses the previous counter level ℓ to transform the current permutation to the next permutation. For example, the successive counter values in (10) correspond to the following sequence of permutations:

$$\begin{aligned} &\overset{[1,1,1]}{\longrightarrow} (1{-}2{-}3{-}\underline{4}) \overset{[1,1,2]}{\longrightarrow} (1{-}2{-}\underline{4}{-}3) \overset{[1,1,3]}{\longrightarrow} (1{-}\underline{4}{-}2{-}3) \overset{[1,1,4]}{\longrightarrow} (4{-}1{-}2{-}\underline{3}) \overset{[1,2,1]}{\longrightarrow} (\underline{4}{-}1{-}3{-}2) \quad (11)\\ &\overset{[1,2,2]}{\longrightarrow} (1{-}\underline{4}{-}3{-}2) \overset{[1,2,3]}{\longrightarrow} (1{-}3{-}\underline{4}{-}2) \overset{[1,2,4]}{\longrightarrow} (1{-}\underline{3}{-}2{-}4) \overset{[1,3,1]}{\longrightarrow} (3{-}1{-}2{-}\underline{4}) \overset{[1,3,2]}{\longrightarrow} \cdots\\ &\overset{[2,3,3]}{\longrightarrow} (1{-}\underline{4}{-}2{-}3) \overset{[2,3,4]}{\longrightarrow} (1{-}2{-}\underline{4}{-}3) \overset{[1,1,1]}{\longrightarrow} (1{-}2{-}3{-}\underline{4}) \longrightarrow \cdots . \end{aligned}$$

To interpret the above, consider a general step of the form

$$\overset{[c_2,c_3,c_4]}{\longrightarrow} \cdots (x_1{-}x_2{-}x_3{-}x_4) \overset{[c'_2,c'_3,c'_4]}{\longrightarrow} (x'_1{-}x'_2{-}x'_3{-}x'_4)\cdots$$

We start with the counter value $[c_2, c_3, c_4]$ and permutation $(x_1{-}x_2{-}x_3{-}x_4)$. After calling `Inc`, the counter is updated to $[c'_2, c'_3, c'_4]$, and it returns the level ℓ of $[c_2, c_3, c_4]$. If $\ell = 1$, we may[5] terminate; otherwise, $\ell \in \{2, 3, 4\}$. We find the index i such that $x_i = \ell$ (for some $i = 1, 2, 3, 4$). UPDATE will then exchange x_i with its neighbor x_{i+1} or x_{i-1}. The resulting permutation is $(x'_1{-}x'_2{-}x'_3{-}x'_4)$.

In (11), we indicate x_i by an underscore, "$\underline{x_i}$". The choice of which neighbor (x_{i-1} or x_{i+1}) depends on whether we are in the "UP" phase or "DOWN" phase of level ℓ. Let $UP[1..n]$ be a Boolean array where $UP[\ell]$ is true in the UP phase, and false in the DOWN phase when we are incrementing a counter at level ℓ. Moreover, the the value of $UP[\ell]$ is changed (flipped) each time $C[\ell]$ is reinitialized to 1. For instance, in the first row of (11), $UP[4] =$ false and so the entry 4 is moving down with each swap involving 4. In the next row, $UP[4] =$ true and so the entry 4 is moving up with each swap.

Hence we modify our previous Inc subroutine to include this update:

[5] In case we want to continue, the case $\ell = 1$ is treated as if $\ell = n$. E.g., in (11), the case $\ell = 1$ is treated as $\ell = 4$.

```
INCREMENT(C)
Output: Increments C, updates UP, and returns the previous level of C
    ℓ ← n.
    while (C[ℓ] = ℓ) ∧ (ℓ > 1)    ◁ Loop to find the counter level
        C[ℓ] ← 1;
        UP[ℓ] ← ¬UP[ℓ];    ◁ Flips the boolean value UP[ℓ]
        ℓ--.
    if (ℓ > 1)
        C[ℓ]++.
    return(ℓ).
```

For a given level ℓ, the UPDATE routine need to find the "position" i where $per[i] = \ell$ $(i = 1, \ldots, n)$. We could search for this position in $O(n)$ time, but it is more efficient to maintain this information directly: let $pos[\ell]$ denote the current position of ℓ. Thus the $pos[1..n]$ is just the inverse of the array $per[1..n]$ in the sense that

$$per[pos[\ell]] = \ell \qquad (\ell = 1, \ldots, n).$$

We can now specify the UPDATE routine to update both *pos* and *per*:

```
UPDATE(ℓ)
    if (UP[ℓ])
        per[pos[ℓ]] ⇔ per[pos[ℓ] + 1];    ◁ modify permutation
        pos[per[pos[ℓ]]] ← pos[ℓ];    ◁ update position array
        pos[ℓ]++;    ◁ update position array
    else
        per[pos[ℓ]] ⇔ per[pos[ℓ] − 1];
        pos[per[pos[ℓ]]] ← pos[ℓ];
        pos[ℓ]--;
```

Thus, the final algorithm is:

```
JOHNSON-TROTTER GENERATOR
Input: natural number n ≥ 2
    ▷ Initialization
        per[1..n] ← [1, 2, ..., n].    ◁ Initial permutation
        pos[1..n] ← [1, 2, ..., n].    ◁ Initial positions
        C[2..n] ← [1, 1, ..., 1].    ◁ Initial counter value
    ▷ Main Loop
        do
            ℓ ← Increment(C);
            UPDATE(ℓ);
            CONSUME(per).
        while(ℓ > 1)
```

REMARKS:
1. In practice, we can introduce early termination criteria into our permutation generator. For instance, in the bin packing application, there is a trivial lower bound on the number of bins, namely $b_0 = \lceil (\sum_{i=1}^{n} w_i)/M \rceil$.

We can stop when we found a solution with b_0 bins. If we want only an approximate optimal, say within a factor of 2, we may exit when the we achieve $\leq 2b_0$ bins.
2. We have focused on permutations of distinct objects. In case the objects may be identical, more efficient techniques may be devised. For more information about permutation generation, see the book of Paige and Wilson [2]. Knuth's much anticipated 4th volume will treat permutations; this will no doubt become a principle reference for the subject.
3. The Java code for the Johnson-Trotter Algorithm is presented in an appendix of this chapter.

Exercises

Exercise 7.1:
(a) Draw the adjacency bigraph corresponding to 4-permutations. HINT: first draw the adjacency graph for 3-permutations and view 4-permutations as extension of 3-permutations.
(b) How many edges are there in the adjacency bigraph of n-permutations?
(c) What is the radius and diameter of the bigraph in part (b)? [See definition of radius and diameter in Exercise 4.8 (Chapter 4).] ♢

Exercise 7.2: Another way to list all the n-permutations in S_n is lexicographic ordering: $(x_1 - \cdots - x_n) < (x'_1 - \cdots - x'_n)$ if the first index i such that $x_i \neq x'_i$ satisfies $x_i < x'_i$. Thus the lexicographic smallest n-permutation is $(1 - 2 - \cdots - n)$. Give an algorithm to generate n-permutations in lexicographic ordering. Compare this algorithm to the Johnson-Trotter algorithm. ♢

Exercise 7.3: All adjacency orderings of 2- and 3-permutations are cyclic. Is it true of 4-permutations? ♢

Exercise 7.4: Two n-permutations π, π' are **cylic equivalent** if $\pi = (x_1 - x_2 - \cdots - x_n)$ and $\pi' = (x_i - x_{i+1} - \cdots - x_n - x_1 - x_2 - \cdots - x_{i-1})$ for some $i = 1, \ldots, n$. A **cyclic n-permutation** is an equivalence class of the cyclic equivalence relation. Note that there are exactly n permutations in each cyclic n-permutation. Let S'_n denote the set of cylic n-permutations. So $|S'_n| = (n-1)!$. Again, we can define the cylic permutation graph G'_n whose vertex set is S'_n, and edges determined by adjacent pairs of cyclic permutations. Give an efficient algorithm to generate a Hamiltonian cycle of G'_n. ♢

End Exercises

§A. APPENDIX: Java Code for Permutations

```
/**************************************************
 * Per(mutations)
 *      This generates the Johnson-Trotter permutation order.
 *      By n-permutation, we mean a permutation of the symbols {1,2,...,n}.
 *
 * Usage:
 *      % javac Per.java
 *      % java Per [n=3] [m=0]
 *
 *      will print all n-permutations.  Default values n=3 and m=0.
 *      If m=1, output in verbose mode.
 *      Thus "java Per" will print
 *            (1,2,3), (1,3,2), (3,1,2), (3,2,1), (2,3,1), (2,1,3).
 *      See Lecture Notes for details of this algorithm.
 *
 **************************************************/

public class Per {

 // Global variables
 ////////////////////////////////////////////////
 static int n;                  // n-permutations are being considered
                      // Quirk: Following arrays are indexed from 1 to n
 static int[] per;              // represents the current n-permutation
 static int[] pos;              // inverse of per:  per[pos[i]]=i (for i=1..n)
 static int[] C;                // Counter array: 1 <= C[i] <= i (for i=1..n)
 static boolean[] UP;           // UP[i]=true iff pos[i] is increasing
                                //     (going up) in the current phase

 // Display permutation or position arrays
 ////////////////////////////////////////////////
 static void showArray(int[] myArray, String message){
   System.out.print(message);
   System.out.print("(" + myArray[1]);
   for (int i=2; i<=n; i++)
     System.out.print("," + myArray[i]);
   System.out.println(")");
 }

 // Print counter
 ////////////////////////////////////////////////
 static void showC(String m){
   System.out.print(m);
   System.out.print("(" + C[2]);
   for (int i=3; i<=n; i++)
     System.out.print("," + C[i]);
   System.out.println(")");
 }

 // Increment counter
 ////////////////////////////////////////////////
 static int inc(){
   int ell=n;
   while ((C[ell]==ell) && (ell>1)){
        UP[ell] = !(UP[ell]);      // flip Boolean flag
```

```
      C[ell--]=1;
  }
  if (ell>1)
      C[ell]++;
  return ell;                    // level of previous counter value
}

// Update per and pos arrays
//////////////////////////////////////////////////
static void update(int ell){
  int tmpSymbol;            // this is not necessary, but for clarity
  if (UP[ell]) {
    tmpSymbol = per[pos[ell]+1];  // Assert: pos[ell]+1 makes sense!
    per[pos[ell]] = tmpSymbol;
    per[pos[ell]+1] = ell;
    pos[ell]++;
    pos[tmpSymbol]--;
  } else {
    tmpSymbol = per[pos[ell]-1];  // Assert: pos[ell]-1 makes sense!
    per[pos[ell]]= tmpSymbol;
    per[pos[ell]-1] = ell;
    pos[ell]--;
    pos[tmpSymbol]++;
  }
}

// Main program
//////////////////////////////////////////////////
public static void main (String[] args)
      throws java.io.IOException
{
  //Command line Processing
  n=3;                    // default value of n
  boolean verbose=false;  // default is false (corresponds to second argument = 0)
  if (args.length>0)
      n = Integer.parseInt(args[0]);
  if ((args.length>1) && (Integer.parseInt(args[1]) != 0))
      verbose = true;

  //Initialize
  per = new int[n+1];
  pos = new int[n+1];
  C   = new int[n+1];
  UP  = new boolean[n+1];
  for (int i=0; i<=n; i++) {
    per[i]=i;
    pos[i]=i;
    C[i]=1;
    UP[i]=false;
  }

  //Setup For Loop
  int count=0;             // only used in verbose mode
  int ell=1;
  System.out.println("Johnson-Trotter ordering of "+ n + "-permutations");
  if (verbose)
      showArray(per, count + ", level="+ ell + " :\t" );
  else
      showArray(per, "");
```

```
   //Main Loop
   do {
     ell = inc();
     update(ell);
     if (verbose)
        count++;
        showArray(per, count + ", level="+ ell + " :\t" );
     else
        showArray(per, "");
   } while (ell>1);

 }//main
}//class Per
```

References

[1] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Comm. of the ACM*, 29(4):320–330, 1986.

[2] E. Page and L. Wilson. *An Introduction to Computational Combinatorics*. Cambridge Computer Science Texts, No. 9. Cambridge University Press, 1979.

[3] T. W. Parsons. Letter: A forgotten generation of permutations, 1977.

[4] R. Sedgewick. Permutation generation methods. *Computing Surveys*, 9(2):137–164, 1977.

[5] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, Philadelphia, PA, 1974.

[6] J. S. Vitter. The design and analysis of dynamic huffman codes. *J. of the ACM*, 34(4):825–845, 1987.