Lecture IV
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. It may seem that this approach is rather limited. Nevertheless, many important problems have special features that allow efficient solution using this approach. The essential point of greedy solutions is that we never have to revise our greedy decisions.

The greedy method is supposed to exemplify the idea of "local search". But closer examination of greedy algorithms will often reveal some global information being used. Such global information is usually minimal, and typically amounts to knowing the sorted values of all the keys in the input set. 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. Huffman Code

We begin with an informally stated problem:

> (P) Given a string s of characters 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 (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 extremely common 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** 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 $C(s)$. In general there is no 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. Our 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 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 $C(d) = 1$.

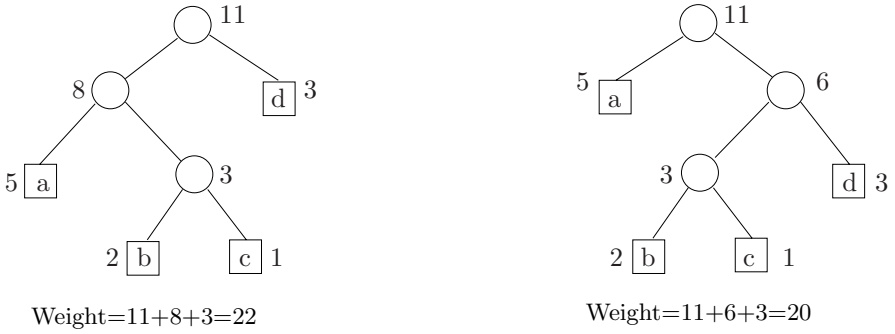

Figure 1: Two prefix-free codes and their code trees.

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

$$COST(f, C) := \sum_{a \in \Sigma} |C(a)| \cdot f(a). \tag{1}$$

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 (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 on Σ, we associate a **frequency** to each node of a code tree T_C: the frequency of a leaf is simply the frequency $f(x)$ of the character x at that leaf, and the frequency of an

internal node is the sum of the frequencies of its children. The frequency of a code tree is the frequency of its root. The **cost** $COST(T_C)$ of the code tree is the sum of the frequencies of all its **internal** nodes. The reader may verify that $COST(f, C) = COST(T_C)$.

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. We now present a greedy algorithm for the 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 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^*.

Implementation and complexity. This algorithm is easily implemented using a priority queue on S. 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 the red-black trees in Lecture III) 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 1 *There is an optimal Huffman code in which the two least frequent characters label some deepest pair.*

Proof. Suppose b, c are two characters at depths $D(b), D(c)$ (respectively) in a code tree T. If we exchange the labels 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':

$$COST(f') = COST(f', C'). \tag{2}$$

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

$$COST(f, C) = COST(f', C') + f(b) + f(b'). \tag{3}$$

It is easily seen from our lemma that

$$COST(f) = COST(f') + f(b) + f(b'). \tag{4}$$

From equations (2), (3) and (4), we conclude $COST(f) = COST(f, C)$, *i.e.*, C is optimal. ■

Remarks: Despite the simplicity of this algorithm, its publication in 1952 by D. A. Huffman was considered a major achievement. This algorithm is clearly useful for compressing binary files. For data compression, there is an important variant of the Huffman coding problem: we seek to construct an optimal code in an "online fashion". That is, the frequency function f is initially identically zero, and at each time step, the frequency of a character is incremented by 1 (this corresponds to reading successive characters in a file). The algorithm must dynamically maintain an optimal code tree for f as it changes in this manner. Such an algorithm is operative in the Unix utility compress/uncompress. There are other related encoding problems. Some data have special "coherence" structures. For instance, a rectangular pixel map has the property that the same pixel value has a tendency to be repeated in its neighborhood. Then it makes sense to encode a "run" of a given pixel value by specifying the length of the run. This is called **run length encoding**.

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 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)	· · · – · –		

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, respectivly. 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 1.1: 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 1.2: 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 1.3: 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 1.4: Suppose you are given the frequencies f_i in sorted order. Show that you can construct the Huffman tree in linear time. ♢

Exercise 1.5: 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 Hoffmann 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 1.6: 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 1.7: Further the above 3-ary Huffman tree construction to arbitrary k-ary codes for $k \geq 4$. ♢

Exercise 1.8: 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 2 for a skew Huffman tree for alphabet $\{a, b, c\}$ and $f(a) = 3$, $f(b) = 1$ and $f(c) = 6$.

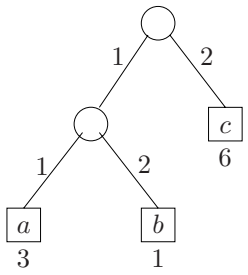

Figure 2: A skew Huffman tree with skew cost of 21.

(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 1.9: (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 1.10: (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 1.11: 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 1.12: (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 1.13: (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. Punction marks and spaces are part of this string. But new lines (CRLF) do not contribute any symbols to the string. 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. Is there a general formulation of these extensions? ♢

End Exercises

§2. Dynamic Huffman Coding

The above Huffman coding method has two deficiencies: (1) It is a 2-pass algorithm in which the first pass over the string s being encoded is to determine the frequency count of symbols in s. This makes the coding unsuitable for realtime data transmissions as well as requiring buffer space. (2) The Huffman code tree must be explicitly transmitted before the decoding can begin. 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 algorithms for this here are the FGK Algorithm (Faller 1973, Gallager 1978, Knuth 1985) and the Lambda Algorithm (Vitter 1987). See [2]. Vitter's algorithm ensures that the transmitted code is $\leq H_2(s) + |s| - 1$ where $H_2(s)$ is the number of bits transmitted by the 2-pass Huffman code for s, independent of alphabet size. It can be shown that FGK transmit at most $2H_2(s) + 4|s|$.

The key idea is the Sibling Property of Gallagher. Let T be a full binary tree on $k \geq 1$ leaves with 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 of T has exactly two children. Thus T has $2k - 1$ nodes. Such a tree T is called a **code tree**. We say T has the **Sibling Property** if its nodes can be numbered from 1 to $2k - 1$ so that (1) if w_i is the weight of node i then $w_i \leq w_{i+1}$ for $i = 1, \ldots, 2k - 2$, and (2) nodes $2j - 1$ and $2j$ (for $j = 1, \ldots, k$) are siblings. Property (2) is non-trivial because the possibility of equal weights meant that the sorting order imposed by (1) is not unique.

We say T is **Huffman** if it can be constructed by the 2-pass Huffman algorithm. Note we view Huffman's algorithm as nondeterministic in this definition, since in the presense of nodes with equal weights, the decision to merge two nodes is not deterministic.

Lemma 2 *T is Huffman iff it has the sibling property.*

Proof. Clearly, if T is Huffman then we can number the nodes in the order that two nodes are merged, and this ordering implies the sibling property. Conversely, the sibling problem prescribes an order for merging pairs of nodes to form a Hoffman tree. **Q.E.D.**

Here is the key problem of dynamic Huffman tree construction. Suppose T is Huffman and we increment by 1 the weight of a single leaf u in T. Let the weights of all the nodes along the path from u to the root is similarly incremented. The result is a code tree T', but it may not be Huffman any more. The key problem is how to restore Huffman-ness.

We consider the following algorithm to restore Huffman-ness. Assume that for each node v in T, $w(v)$ is its weight and $n(v)$ is its position in an ordering of the nodes that satisfy the Sibling Property. Let u be the current node. We now iterate the following process:

```
Restore (u)
    While u is not the root do
        Find the node v with the largest value of n(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 swap is
            really a swap of the entire subtree at u and v.}
        Increment w(u)++ and let u = parent(u). Note that
            u is now the former parent of v!. Break.
    Increment w(u)++. {u is the root}
```

We claim that this algorithm restores T into a Huffman tree in which the weight of the original node u is now incremented.

Let us consider a simple example of how restores work.

We now obtain an encoding for an arbitrary string as follows. Maintain a dynamic Huffman code tree. Make sure that there is one node with weight 0 (call this the 0-node). This node does not represent any letters of the alphabet, but in some sense represents all the yet unseen letters. To process a next character x in the input string, we first check if x is represented in the current code tree. If so, we transmit the current code word for x and increment its frequency count in the current Huffman tree. The tree is restored by our above algorithm. Suppose x is a new character. We transmit the code word for the current 0-node, followed by the canonical representation for x (e.g., the ASCII code for x, if our original character set is in ASCII). Then we expand the 0-node to have two children which are 0-nodes. Now let the right sibling of two 0-nodes to represent the character x. Increment the frequency of this 0-node to 1. We then restore the Huffman tree using the above algorithm.

Decoding is also relatively easy. Each time we receive a sequence of symbols, we know whether it is a new character or a previously encountered one. In either case, we know how to update the Huffman code tree.

Exercises

Exercise 2.1: Give an efficient implementation of the dynamic Huffman code. ♢

Exercise 2.2: 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 2.3: 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 2.4: Consider 3-ary Huffman tree code. State and ♢

Exercise 2.5: Consider 3-ary Huffman tree code. State and prove the Sibling property for this code. ♢

End Exercises

§3. 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 3 the sets $A = \{ab, ac, ad\}$ and $B = \{bc, ca, ad, de\}$ are acyclic. Then the exchange property is witnessed by the edge de.

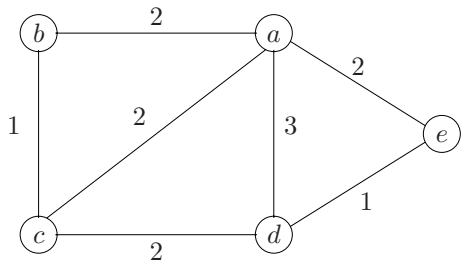

Figure 3: A bigraph with edge costs.

Matroids. The above system (S, I) is called the **graphic matroid** corresponding to graph $G = (V, S)$. In general, a **matroid** is a 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. 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[1] 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.

[1]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 $r_M(R)$.

Fundamental 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[2] 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 3 (Correctness) *Suppose the elements of A are put into A in this order:*

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

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

where $m = |A|$. Let $A_i = \{z_1, z_2, \ldots, z_i\}$, $i = 1, \ldots, m$. Then:
1. A is a base.
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 $(x_1, \ldots, x_m, \ldots, x_n)$.

Remark: 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.

EXERCISES

Exercise 3.1: Consider the graphic matroid in figure 3. Determine its rank function. ♢

Exercise 3.2: The text described a modification of the Greedy Maximum Base Algorithm so that it will solve the MIS problem. Verify its correctness. ♢

Exercise 3.3:
(a) Interpret the induced and contracted submatroids $M|R$ and $M \wedge R$ in the bigraph of figure 3, 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 3.4: 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 3.5: (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 3.6:
(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 3.7: 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 3.8: 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 3.9: 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

§4. 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 graph 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 3 with vertices $V = \{a, b, c, d, e\}$. One such MST is $\{bc, de, ac, ae\}$, with cost 6. It is easy to verify that there are six MST's, as shown in figure 4.

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: reorder the m edges of the input G so that

$$C(e_1) \leq C(e_2) \leq \cdots \leq C(e_m) \tag{5}$$

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:

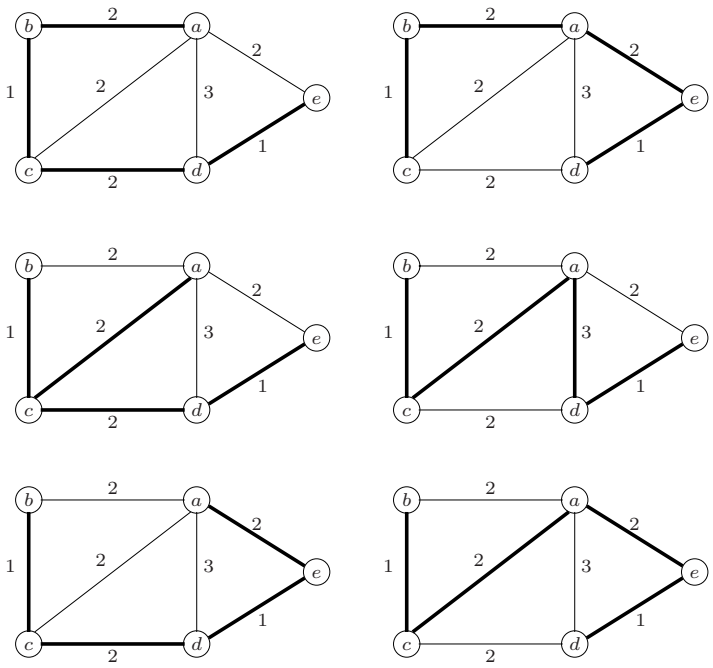

Figure 4: MST's of a bigraph.

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 $n - 1$ 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 algorithms of Boruvka and Prim. But, by definition, Prim-safe implies Boruvka-safe. Hence it is sufficient to prove:

Lemma 4 (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 $e \notin T'$.

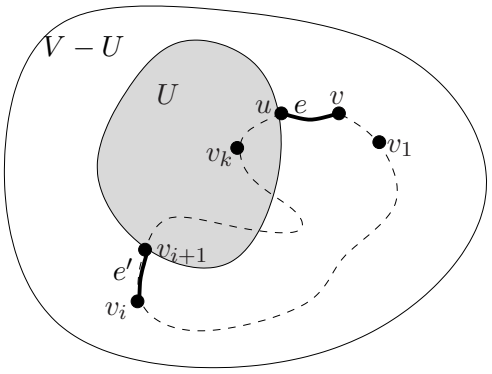

Figure 5: Extending a component U by $e = (u, v)$.

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

$$Z := (u, v, v_1, v_2, \ldots, v_k, u).$$

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

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

(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. Such details in the case of Prim's algorithm is taken up in Lecture V (amortization techniques). Similarly, we will show how to implement Kruskal's algorithm in Lecture XII when we study the union-find data structure.

Safe sets of vertices. For later applications, we inject another definition now. 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=simply, Kruskal, Boruvka or Prim. 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.

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 [1] for further references. Both Boruvkaand Jarnik's work are in Czech. The Prim-Jarník agorithm is very similar in structure to Dijkstra's algorithm which we will encounter in the chapter on minimum cost paths.

Exercises

Exercise 4.1: 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 $G = G_{12}$ be the graph in figure 6. The value $C(v)$ is written next to the node v. For instance

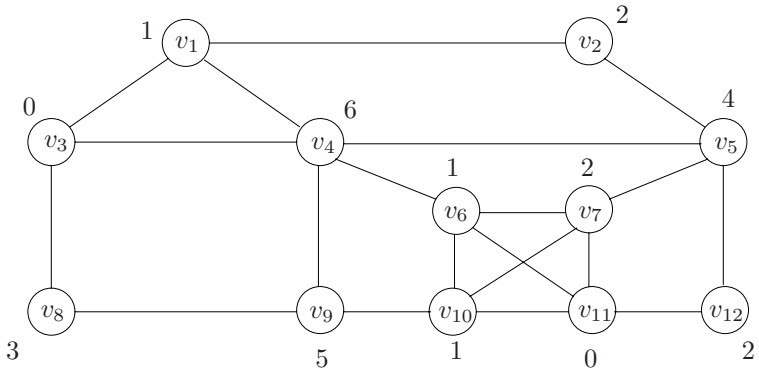

Figure 6: The graph G_{12}.

$C(v_4) = 6$ and $C(v_1, v_4) = 1 + 6 = 7$. Compute an MST of G_{12} using Kruskal's algorithm. Please organize your computation so that we can verify intermediate results. Also state the cost of your minimum spanning tree.
(b) 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 4.2: 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 4.3: Prove that the minimal spanning tree T of an undirected graph G with distinct weights must contain that edge of smallest weight. Must it contain the edge of second smallest weight? Must it contain the edge of third smallest weight? ♢

Exercise 4.4: 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 4.5: Extend the algorithm to finding MIS in contracted matroids. ♢

Exercise 4.6: 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 4.7:
(a) Enumerate the X-safe sets of vertices in figure 3. Here, X is 'simply', 'Kruskal', 'Boruvka' or 'Prim'.
(b) Characterize the safe singletons (relative to any of the three notions of safety). ♢

Exercise 4.8: (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 4.9: 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 4.10: 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 4.11: Is there an analogue of Prim and Boruvka's algorithm for the MIS problem for matroids? ♢

Exercise 4.12: 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 4.13: Student 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?

♢

End Exercises

References

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

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