UNIT-3
GREEDY METHOD

Greedy Algorithms: In many optimization algorithms a series of selections need to be made. In dynamic programming we saw one way to make these selections. Namely, the optimal solution is described in a recursive manner, and then is computed “bottom-up”. Dynamic programming is a powerful technique, but it often leads to algorithms with higher than desired running times. Today we will consider an alternative design technique, called *greedy algorithms*. This method typically leads to simpler and faster algorithms, but it is not as powerful or as widely applicable as dynamic programming. We will give some examples of problems that can be solved by greedy algorithms. (Later in the semester, we will see that this technique can be applied to a number of graph problems as well.) Even when greedy algorithms do not produce the optimal solution, they often provide fast heuristics (nonoptimal solution strategies), are often used in finding good approximations.

Activity Scheduling: *Activity scheduling* and it is a very simple scheduling problem. We are given a set $S = \{1, 2, \ldots, n\}$ of n *activities* that are to be scheduled to use some resource, where each activity must be started at a given start time s_i and ends at a given finish time f_i. For example, these might be lectures that are to be given in a lecture hall, where the lecture times have been set up in advance, or requests for boats to use a repair facility while they are in port.

Because there is only one resource, and some start and finish times may overlap (and two lectures cannot be given in the same room at the same time), not all the requests can be honored. We say that two activities i and j are *noninterfering* if their start-finish intervals do not overlap, more formally, $[s_i, f_i) \cap [s_j, f_j) = \emptyset$. (Note that making the intervals *half open*, two consecutive activities are not considered to interfere.) The *activity scheduling problem* is to select a maximum-size set of mutually noninterfering activities for use of the resource. (Notice that goal here is maximum number of activities, not maximum utilization. Of course different criteria could be considered, but the greedy approach may not be optimal in general.)

How do we schedule the largest number of activities on the resource? Intuitively, we do not like long activities, because they occupy the resource and keep us from honoring other requests. This suggests the following greedy strategy: repeatedly select the activity with the smallest duration $(f_i - s_i)$ and schedule it, provided that it does not interfere with any previously scheduled activities. Although this seems like a reasonable strategy, this turns out to be nonoptimal. (See Problem 17.1-4 in CLRS). Sometimes the design of a correct greedy algorithm requires trying a few different strategies, until hitting on one that works.

Here is a greedy strategy that does work. The intuition is the same. Since we do not like activities that take a long time, let us select the activity that finishes first and schedule it. Then, we skip all activities that interfere with this one, and schedule the next one that has the earliest finish time, and so on. To make the selection process faster, we assume that the activities have been sorted by their finish times, that is,

$$f_1 \le f_2 \le \ldots \le f_n,$$

Assuming this sorting, the pseudocode for the rest of the algorithm is presented below. The output is the list A of scheduled activities. The variable *prev* holds the index of the most recently scheduled activity at any time, in order to determine interferences.

Greedy Activity Scheduler

```
schedule(s[1..n], f[1..n]) {                // given start and finish times
    // we assume f[1..n] already sorted
    List A = <1>                            // schedule activity 1 first
    prev = 1
    for i = 2 to n
        if (s[i] >= f[prev]) {              // no interference?
            append i to A;  prev = i        // schedule i next
```

```
        }
    return A
}
```

It is clear that the algorithm is quite simple and efficient. The most costly activity is that of sorting the activities by finish time, so the total running time is $\Theta(n \log n)$. Fig. 14 shows an example. Each activity is represented by its start-finish time interval. Observe that the intervals are sorted by finish time. Event 1 is scheduled first. It interferes with activity 2 and 3. Then Event 4 is scheduled. It interferes with activity 5 and 6. Finally, activity 7 is scheduled, and it intereferes with the remaining activity. The final output is $\{1, 4, 7\}$. Note that this is not the only optimal schedule. $\{2, 4, 7\}$ is also optimal.

Proof of Optimality: Our proof of optimality is based on showing that the first choice made by the algorithm is the best possible, and then using induction to show that the rest of the choices result in an optimal schedule. Proofs of optimality for greedy algorithms follow a similar structure. Suppose that you have any nongreedy solution.

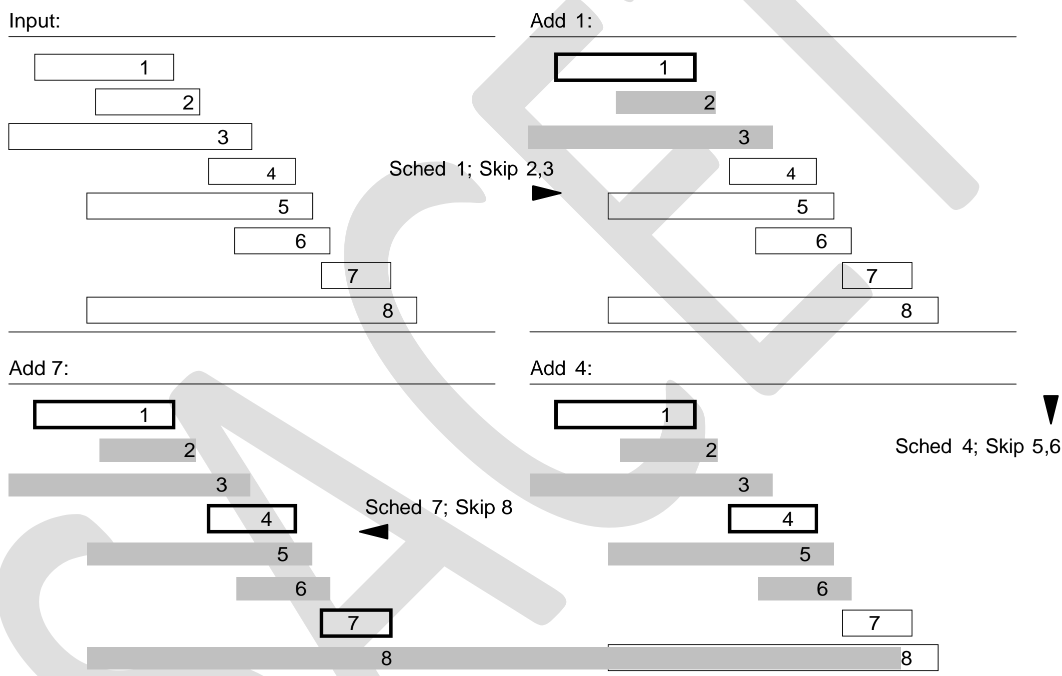


Fig. 14: An example of the greedy algorithm for activity scheduling. The final schedule is $\{1, 4, 7\}$.

Show that its cost can be reduced by being "greedier" at some point in the solution. This proof is complicated a bit by the fact that there may be multiple solutions. Our approach is to show that any schedule that is not greedy can be made more greedy, without decreasing the number of activities.

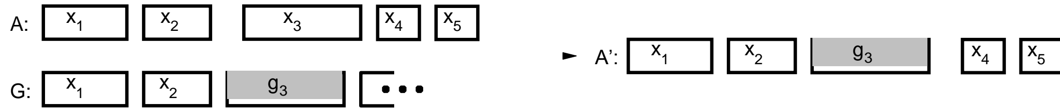
Claim: The greedy algorithm gives an optimal solution to the activity scheduling problem.

Proof: Consider any optimal schedule A that is not the greedy schedule. We will construct a new optimal schedule A' that is in some sense "greedier" than A. Order the activities in increasing order of finish time. Let $A = \langle x_1, x_2, \ldots, x_k \rangle$ be the activities of A. Since A is not the same as the greedy schedule, consider the first activity x_j where these two schedules differ. That is, the greedy schedule is of the form $G = \langle x_1, x_2, \ldots, x_{j-1}, g_j, \ldots \rangle$ where $g_j = x_j$. (Note that $k \geq j$, since otherwise G would have more activities than the optimal schedule, which would be a contradiction.) The greedy algorithm selects the activity with the earliest finish time that does not conflict with any earlier activity. Thus, we know that g_j

does not conflict with any earlier activity, and it finishes before x_j.

Consider the modified “greedier” schedule A' that results by replacing x_j with g_j in the schedule A. (See Fig. 15.) That is,

$$A' = \langle x_1, x_2, \ldots, x_{j-1}, g_j, x_{j+1}, \ldots, x_k \rangle.$$



SACET

Fig.: Proof of optimality for the greedy schedule ($j = 3$).

This is a feasible schedule. (Since g_j cannot conflict with the earlier activities, and it does not conflict with later activities, because it finishes before x_j.) It has the same number of activities as A, and therefore A'

is also optimal. By repeating this process, we will eventually convert A into G, without decreasing the number of activities. Therefore, G is also optimal.

Fractional Knapsack Problem: The classical *(0-1) knapsack problem* is a famous optimization problem. A thief is robbing a store, and finds n items which can be taken. The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but has a knapsack that can only carry W total pounds. Which items should he take? (The reason that this is called 0-1 knapsack is that each item must be left (0) or taken entirely (1). It is not possible to take a fraction of an item or multiple copies of an item.) This optimization problem arises in industrial packing applications. For example, you may want to ship some subset of items on a truck of limited capacity.

In contrast, in the *fractional knapsack problem* the setup is exactly the same, but the thief is allowed to take any *fraction* of an item for a fraction of the weight and a fraction of the value. So, you might think of each object as being a sack of gold, which you can partially empty out before taking.

The 0-1 knapsack problem is hard to solve, and in fact it is an NP-complete problem (meaning that there probably doesn't exist an efficient solution). However, there is a very simple and efficient greedy algorithm for the fractional knapsack problem.

As in the case of the other greedy algorithms we have seen, the idea is to find the right order in which to process items. Intuitively, it is good to have high value and bad to have high weight. This suggests that we first sort the items according to some function that is an decreases with value and increases with weight. There are a few choices that you might try here, but only one works. Let $\rho_i = v_i/w_i$ denote the *value-per-pound ratio*. We sort the items in decreasing order of ρ_i, and add them in this order. If the item fits, we take it all. At some point there is an item that does not fit in the remaining space. We take as much of this item as possible, thus filling the knapsack entirely. This is illustrated in Fig. 16

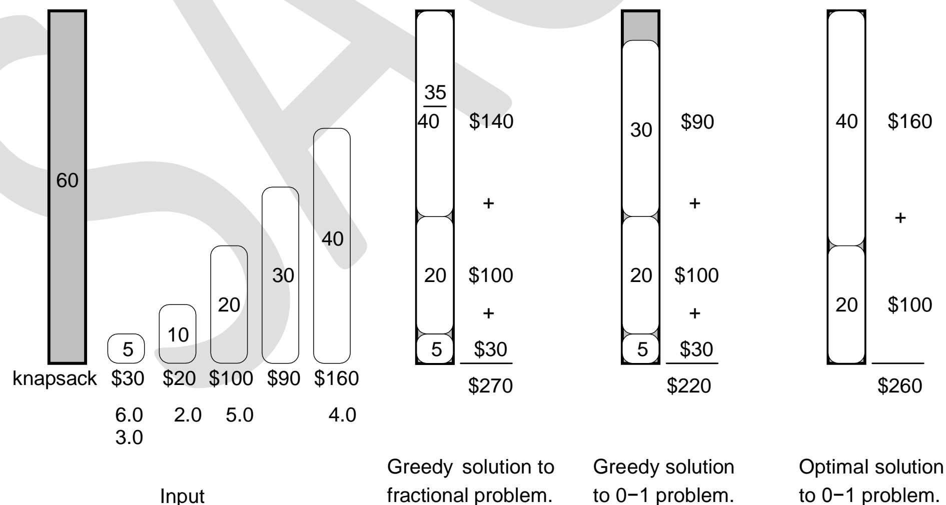


Fig. Example for the fractional knapsack problem.

Correctness: It is intuitively easy to see that the greedy algorithm is optimal for the fractional problem. Given a room with sacks of gold, silver, and bronze, you would obviously take as much gold as possible, then take as much

silver as possible, and then as much bronze as possible. But it would never benefit you to take a little less gold so that you could replace it with an equal volume of bronze.

More formally, suppose to the contrary that the greedy algorithm is not optimal. This would mean that there is an alternate selection that is optimal. Sort the items of the alternate selection in decreasing order by ρ values. Consider the first item i on which the two selections differ. By definition, greedy takes a greater amount of item i than the alternate (because the greedy always takes as much as it can). Let us say that greedy takes x more

SACET

units of object i than the alternate does. All the subsequent elements of the alternate selection are of lesser value than v_i. By replacing x units of any such items with x units of item i, we would increase the overall value of the alternate selection. However, this implies that the alternate selection is not optimal, a contradiction.

Nonoptimality for the 0-1 Knapsack: Next we show that the greedy algorithm is not generally optimal in the 0-1 knapsack problem. Consider the example shown in Fig. 16. If you were to sort the items by ρ_i, then you would first take the items of weight 5, then 20, and then (since the item of weight 40 does not fit) you would settle for the item of weight 30, for a total value of $\$30 + \$100 + \$90 = \$220$. On the other hand, if you had been less greedy, and ignored the item of weight 5, then you could take the items of weights 20 and 40 for a total value of $\$100 + \$160 = \$260$. This feature of "delaying gratification" in order to come up with a better overall solution is your indication that the greedy solution is not optimal.

Lecture 8: Greedy Algorithms: Huffman Coding

Read: Section 16.3 in CLRS.

Huffman Codes: Huffman codes provide a method of encoding data efficiently. Normally when characters are coded using standard codes like ASCII, each character is represented by a fixed-length *codeword* of bits (e.g. 8 bits per character). Fixed-length codes are popular, because its is very easy to break a string up into its individual characters, and to access individual characters and substrings by direct indexing. However, fixed-length codes may not be the most efficient from the perspective of minimizing the total quantity of data.

Consider the following example. Suppose that we want to encode strings over the (rather limited) 4-character alphabet $C = \{a, b, c, d\}$. We could use the following fixed-length code:

Character	a	b	c	d
Fixed-Length Codeword	00	01	10	11

A string such as "abacdaacac" would be encoded by replacing each of its characters by the corresponding binary codeword.

a	b	a	c	d	a	a	c	a	c
00	01	00	10	11	00	00	10	00	10

The final 20-character binary string would be "00010010110000100010".

Now, suppose that you knew the relative probabilities of characters in advance. (This might happen by analyzing many strings over a long period of time. In applications like data compression, where you want to encode one file, you can just scan the file and determine the exact frequencies of all the characters.) You can use this knowledge to encode strings differently. Frequently occurring characters are encoded using fewer bits and less frequent characters are encoded using more bits. For example, suppose that characters are expected to occur with the following probabilities. We could design a *variable-length code* which would do a better job.

Character	a	b	c	d
Probability	0.60	0.05	0.30	0.05
Variable-Length Codeword	0	110	10	111

Notice that there is no requirement that the alphabetical order of character correspond to any sort of ordering applied to the codewords. Now, the same string would be encoded as follows.

a	b	a	c	d	a	a	c	a	c
0	110	0	10	111	0	0	10	0	10

Thus, the resulting 17-character string would be "01100101110010010". Thus, we have achieved a savings of 3 characters, by using this alternative code. More generally, what would be the expected savings for a string of length n? For the 2-bit fixed-length code, the length of the encoded string is just $2n$ bits. For the variable-length code, the expected length of a single encoded character is equal to the sum of code lengths times the respective probabilities of their occurrences. The expected encoded string length is just n times the expected encoded character length.

$$n(0.60 \cdot 1 + 0.05 \cdot 3 + 0.30 \cdot 2 + 0.05 \cdot 3) = n(0.60 + 0.15 + 0.60 + 0.15) = 1.5n.$$

Thus, this would represent a 25% savings in expected encoding length. The question that we will consider today is how to form the best code, assuming that the probabilities of character occurrences are known.

Prefix Codes: One issue that we didn't consider in the example above is whether we will be able to *decode* the string, once encoded. In fact, this code was chosen quite carefully. Suppose that instead of coding the character 'a' as 0, we had encoded it as 1. Now, the encoded string "111" is ambiguous. It might be "d" and it might be "aaa". How can we avoid this sort of ambiguity? You might suggest that we add separation markers between the encoded characters, but this will tend to lengthen the encoding, which is undesirable. Instead, we would like the code to have the property that it can be uniquely decoded.

Note that in both the variable-length codes given in the example above no codeword is a *prefix* of another. This turns out to be the key property. Observe that if two codewords did share a common prefix, e.g. $a \rightarrow 001$ and $b \rightarrow 00101$, then when we see $00101\ldots$ how do we know whether the first character of the encoded message is a or b. Conversely, if no codeword is a prefix of any other, then as soon as we see a codeword appearing as a prefix in the encoded text, then we know that we may decode this without fear of it matching some longer codeword. Thus we have the following definition.

Prefix Code: An assignment of codewords to characters so that no codeword is a prefix of any other.

Observe that any binary prefix coding can be described by a binary tree in which the codewords are the leaves of the tree, and where a left branch means "0" and a right branch means "1". The code given earlier is shown in the following figure. The length of a codeword is just its depth in the tree. The code given earlier is a prefix code, and its corresponding tree is shown in the following figure.

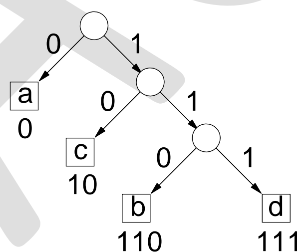


Fig.: Prefix codes.

Decoding a prefix code is simple. We just traverse the tree from root to leaf, letting the input character tell us which branch to take. On reaching a leaf, we output the corresponding character, and return to the root to continue the process.

Expected encoding length: Once we know the probabilities of the various characters, we can determine the total length of the encoded text. Let $p(x)$ denote the probability of seeing character x, and let $d_T(x)$ denote the length of the codeword (depth in the tree) relative to some prefix tree T. The expected number of bits needed to encode a text with n characters is given in the following formula:

$$B(T) = n \sum_{x \in C} p(x) d_T(x).$$

SACET

This suggests the following problem:

Optimal Code Generation: Given an alphabet C and the probabilities $p(x)$ of occurrence for each character $x \in C$, compute a prefix code T that minimizes the expected length of the encoded bit-string, $B(T)$.

Note that the optimal code is not unique. For example, we could have complemented all of the bits in our earlier code without altering the expected encoded string length. There is a very simple algorithm for finding such a code. It was invented in the mid 1950's by David Huffman, and is called a *Huffman code.*. By the way, this code is used by the Unix utility pack for file compression. (There are better compression methods however. For example, compress, gzip and many others are based on a more sophisticated method called the *Lempel-Ziv coding*.)

Huffman's Algorithm: Here is the intuition behind the algorithm. Recall that we are given the occurrence probabilities for the characters. We are going to build the tree up from the leaf level. We will take two characters x and y, and "merge" them into a single *super-character* called z, which then replaces x and y in the alphabet. The character z will have a probability equal to the sum of x and y's probabilities. Then we continue recursively building the code on the new alphabet, which has one fewer character. When the process is completed, we know the code for z, say 010. Then, we append a 0 and 1 to this codeword, given 0100 for x and 0101 for y.

Another way to think of this, is that we merge x and y as the left and right children of a root node called z. Then the subtree for z replaces x and y in the list of characters. We repeat this process until only one super-character remains. The resulting tree is the final prefix tree. Since x and y will appear at the bottom of the tree, it seem most logical to select the two characters with the smallest probabilities to perform the operation on. The result is Huffman's algorithm. It is illustrated in the following figure.

The pseudocode for Huffman's algorithm is given below. Let C denote the set of characters. Each character $x \in C$ is associated with an occurrence probability $x.prob$. Initially, the characters are all stored in a *priority queue* Q. Recall that this data structure can be built initially in $O(n)$ time, and we can extract the element with the smallest key in $O(\log n)$ time and insert a new element in $O(\log n)$ time. The objects in Q are sorted by probability. Note that with each execution of the for-loop, the number of items in the queue decreases by one. So, after $n-1$ iterations, there is exactly one element left in the queue, and this is the root of the final prefix code tree.

Correctness: The big question that remains is why is this algorithm correct? Recall that the cost of any encoding tree T is $B(T) = \sum_x p(x) d_T(x)$. Our approach will be to show that any tree that differs from the one constructed by Huffman's algorithm can be converted into one that is equal to Huffman's tree without increasing its cost. First, observe that the Huffman tree is a *full binary tree*, meaning that every internal node has exactly two children. It would never pay to have an internal node with only one child (since such a node could be deleted), so we may limit consideration to full binary trees.

Claim: Consider the two characters, x and y with the smallest probabilities. Then there is an optimal code tree in which these two characters are siblings at the maximum depth in the tree.

Proof: Let T be any optimal prefix code tree, and let b and c be two siblings at the maximum depth of the tree. Assume without loss of generality that $p(b) \le p(c)$ and $p(x) \le p(y)$ (if this is not true, then rename these characters). Now, since x and y have the two smallest probabilities it follows that $p(x) \le p(b)$ and $p(y) \le p(c)$. (In both cases they may be equal.) Because b and c are at the deepest level of the tree we know that $d(b) \ge d(x)$ and $d(c) \ge d(y)$. (Again, they may be equal.) Thus, we have $p(b) - p(x) \ge 0$ and $d(b) - d(x) \ge 0$, and hence their product is nonnegative. Now switch the positions of x and b in the tree, resulting in a new tree T'. This is illustrated in the following figure.

Next let us see how the cost changes as we go from T to T'. Almost all the nodes contribute the same to the expected cost. The only exception are nodes x and b. By subtracting the old contributions of these

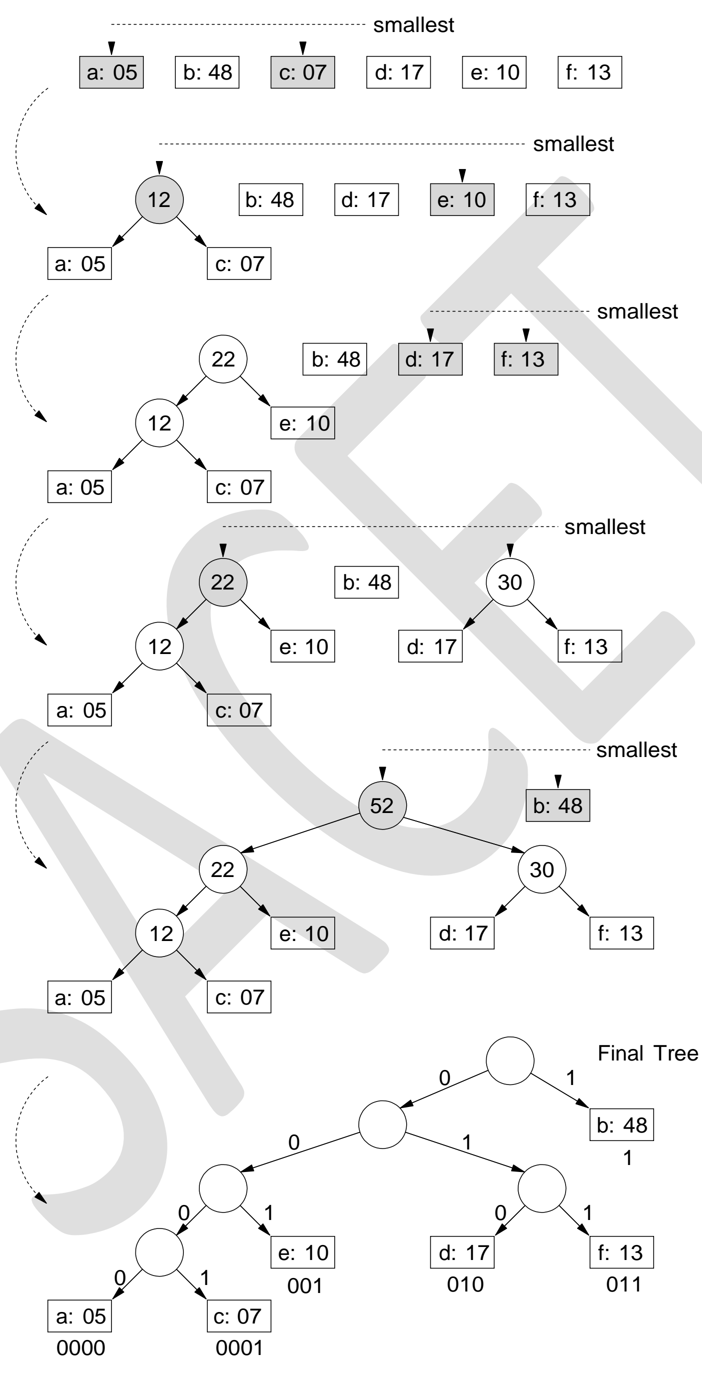


Fig: Huffman's Algorithm.

```
Huffman(int n, character C[1..n]) {
    Q = C;                              // priority queue
    for i = 1 to n-1 {
        z = new internal tree node;
        z.left = x = Q.extractMin();    // extract smallest probabilities
        z.right = y = Q.extractMin();
        z.prob = x.prob + y.prob;       // z's probability is their sum
        Q.insert(z);                    // insert z into queue
    }
    return the last element left in Q as the root;
}
```

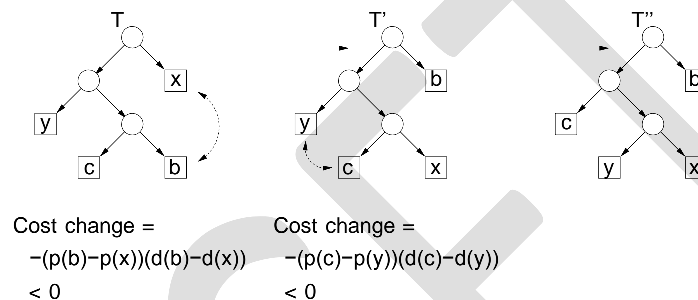


Fig.: Correctness of Huffman's Algorithm.

nodes and adding in the new contributions we have

$$\begin{aligned} B(T') &= B(T) - p(x)d(x) + p(x)d(b) - p(b)d(b) + p(b)d(x) \\ &= B(T) + p(x)(d(b) - d(x)) - p(b)(d(b) - d(x)) \\ &= B(T) - (p(b) - p(x))(d(b) - d(x)) \\ &\leq B(T) \qquad \text{because } (p(b) - p(x))(d(b) - d(x)) \geq 0. \end{aligned}$$

Thus the cost does not increase, implying that T' is an optimal tree. By switching y with c we get a new tree T'', which by a similar argument is also optimal. The final tree T'' satisfies the statement of the claim.

The above theorem asserts that the first step of Huffman's algorithm is essentially the proper one to perform. The complete proof of correctness for Huffman's algorithm follows by induction on n (since with each step, we eliminate exactly one character).

Claim: Huffman's algorithm produces the optimal prefix code tree.

Proof: The proof is by induction on n, the number of characters. For the basis case, $n = 1$, the tree consists of a single leaf node, which is obviously optimal.

Assume inductively that when strictly fewer than n characters, Huffman's algorithm is guaranteed to produce the optimal tree. We want to show it is true with exactly n characters. Suppose we have exactly n characters. The previous claim states that we may assume that in the optimal tree, the two characters of lowest probability x and y will be siblings at the lowest level of the tree. Remove x and y, replacing them with a new character z whose probability is $p(z) = p(x) + p(y)$. Thus $n - 1$ characters remain.

Consider any prefix code tree T made with this new set of $n - 1$ characters. We can convert it into a prefix code tree T' for the original set of characters by undoing the previous operation and replacing z with x

and y (adding a "0" bit for x and a "1" bit for y). The cost of the new tree is

$$\begin{aligned} B(T') &= B(T) - p(z)d(z) + p(x)(d(z)+1) + p(y)(d(z)+1) \\ &= B(T) - (p(x)+p(y))d(z) + (p(x)+p(y))(d(z)+1) \\ &= B(T) + (p(x)+p(y))(d(z)+1-d(z)) \\ &= B(T) + p(x) + p(y). \end{aligned}$$

Since the change in cost depends in no way on the structure of the tree T, to minimize the cost of the final tree T', we need to build the tree T on $n-1$ characters optimally. By induction, this exactly what Huffman's algorithm does. Thus the final tree is optimal.

Minimum Spanning Trees: A common problem in communications networks and circuit design is that of connecting together a set of nodes (communication sites or circuit components) by a network of minimal total length (where length is the sum of the lengths of connecting wires). We assume that the network is undirected. To minimize the length of the connecting network, it never pays to have any cycles (since we could break any cycle without destroying connectivity and decrease the total length). Since the resulting connection graph is connected, undirected, and acyclic, it is a *free tree*.

The computational problem is called the *minimum spanning tree* problem (MST for short). More formally, given a connected, undirected graph $G = (V, E)$, a *spanning tree* is an acyclic subset of edges $T \subseteq E$ that connects all the vertices together. Assuming that each edge (u, v) of G has a numeric weight or cost, $w(u, v)$, (may be zero or negative) we define the cost of a spanning tree T to be the sum of edges in the spanning tree

$$w(T) = \sum_{(u,v)\in T} w(u, v).$$

A *minimum spanning tree* (MST) is a spanning tree of minimum weight. Note that the minimum spanning tree may not be unique, but it is true that if all the edge weights are distinct, then the MST will be distinct (this is a rather subtle fact, which we will not prove). Fig. 31 shows three spanning trees for the same graph, where the shaded rectangles indicate the edges in the spanning tree. The one on the left is not a minimum spanning tree, and the other two are. (An interesting observation is that not only do the edges sum to the same value, but in fact the same set of edge weights appear in the two MST's. Is this a coincidence? We'll see later.)

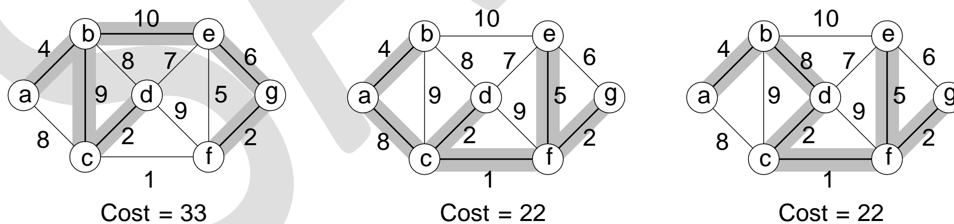


Fig. 31: Spanning trees (the middle and right are minimum spanning trees.

Steiner Minimum Trees: Minimum spanning trees are actually mentioned in the U.S. legal code. The reason is that AT&T was a government supported monopoly at one time, and was responsible for handling all telephone connections. If a company wanted to connect a collection of installations by an private internal phone system,

AT&T was required (by law) to connect them in the minimum cost manner, which is clearly a spanning tree . . . or is it?

Some companies discovered that they could actually reduce their connection costs by opening a new bogus installation. Such an installation served no purpose other than to act as an intermediate point for connections.

An example is shown in Fig. 32. On the left, consider four installations that lie at the corners of a 1×1 square. Assume that all edge lengths are just Euclidean distances. It is easy to see that the cost of any MST for this configuration is 3 (as shown on the left). However, if you introduce a new installation at the center, whose distance to each of the other four points is $1/\sqrt{2}$. It is now possible to connect these five points with a total cost of $4/\sqrt{2} = 2\sqrt{2} \approx 2.83$. This is better than the MST.

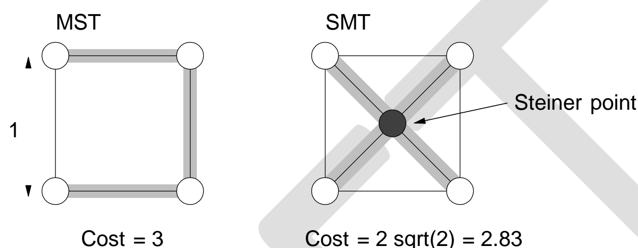


Fig. 32: Steiner Minimum tree.

In general, the problem of determining the lowest cost interconnection tree between a given set of nodes, assuming that you are allowed additional nodes (called *Steiner points*) is called the *Steiner minimum tree* (or SMT for short). An interesting fact is that although there is a simple greedy algorithm for MST's (as we will see below), the SMT problem is much harder, and in fact is NP-hard. (Luckily for AT&T, the US Legal code is rather ambiguous on the point as to whether the phone company was required to use MST's or SMT's in making connections.)

Generic approach: We will present two *greedy* algorithms (Kruskal's and Prim's algorithms) for computing a minimum spanning tree. Recall that a *greedy algorithm* is one that builds a solution by repeated selecting the cheapest (or generally locally optimal choice) among all options at each stage. An important characteristic of greedy algorithms is that once they make a choice, they never "unmake" this choice. Before presenting these algorithms, let us review some basic facts about free trees. They are all quite easy to prove.

Lemma:

- A free tree with n vertices has exactly $n - 1$ edges.
- There exists a unique path between any two vertices of a free tree.
- Adding any edge to a free tree creates a unique cycle. Breaking *any* edge on this cycle restores a free tree.

Let $G = (V, E)$ be an undirected, connected graph whose edges have numeric edge weights (which may be positive, negative or zero). The intuition behind the greedy MST algorithms is simple, we maintain a subset of edges A, which will initially be empty, and we will add edges one at a time, until A equals the MST. We say that a subset $A \subseteq E$ is *viable* if A is a subset of edges in some MST. (We cannot say "the" MST, since it is not necessarily unique.) We say that an edge $(u, v) \in E - A$ is *safe* if $A \cup \{(u, v)\}$ is viable. In other words, the choice (u, v) is a safe choice to add so that A can still be extended to form an MST. Note that if A is viable it cannot contain a cycle. A generic greedy algorithm operates by repeatedly adding any *safe* edge to the current spanning tree. (Note that viability is a property of subsets of edges and safety is a property of a single edge.)

When is an edge safe? We consider the theoretical issues behind determining whether an edge is safe or not. Let S be a subset of the vertices $S \subseteq V$. A *cut* $(S, V - S)$ is just a partition of the vertices into two disjoint subsets. An edge (u, v) *crosses* the cut if one endpoint is in S and the other is in $V - S$. Given a subset of edges A, we

say that a cut *respects* A if no edge in A crosses the cut. It is not hard to see why respecting cuts are important to this problem. If we have computed a partial MST, and we wish to know which edges can be added that do *not* induce a cycle in the current MST, any edge that crosses a respecting cut is a possible candidate.

An edge of E is a *light edge* crossing a cut, if among all edges crossing the cut, it has the minimum weight (the light edge may not be unique if there are duplicate edge weights). Intuition says that since all the edges that cross a respecting cut do not induce a cycle, then the lightest edge crossing a cut is a natural choice. The main theorem which drives both algorithms is the following. It essentially says that we can always augment A by adding the minimum weight edge that crosses a cut which respects A. (It is stated in complete generality, so that it can be applied to both algorithms.)

MST Lemma: Let $G = (V, E)$ be a connected, undirected graph with real-valued weights on the edges. Let A be a viable subset of E (i.e. a subset of some MST), let $(S, V - S)$ be any cut that respects A, and let (u, v) be a light edge crossing this cut. Then the edge (u, v) is *safe* for A.

Proof: It will simplify the proof to assume that all the edge weights are distinct. Let T be any MST for G (see Fig.). If T contains (u, v) then we are done. Suppose that no MST contains (u, v). We will derive a contradiction.

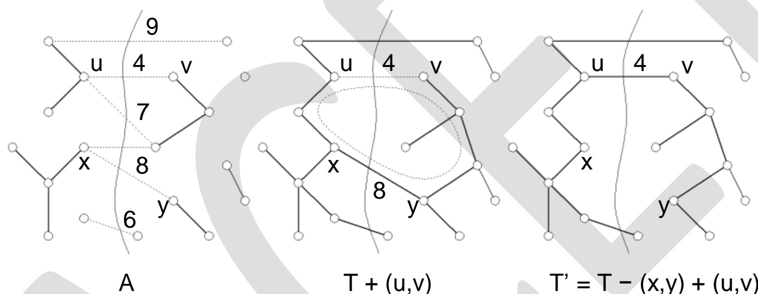


Fig. 33: Proof of the MST Lemma. Edge (u, v) is the light edge crossing cut $(S, V - S)$.

Add the edge (u, v) to T, thus creating a cycle. Since u and v are on opposite sides of the cut, and since any cycle must cross the cut an even number of times, there must be at least one other edge (x, y) in T that crosses the cut.

The edge (x, y) is not in A (because the cut respects A). By removing (x, y) we restore a spanning tree, call it T'. We have

$$w(T') = w(T) - w(x, y) + w(u, v).$$

Since (u, v) is lightest edge crossing the cut, we have $w(u, v) < w(x, y)$. Thus $w(T') < w(T)$. This contradicts the assumption that T was an MST.

Kruskal's Algorithm: Kruskal's algorithm works by attempting to add edges to the A in increasing order of weight (lightest edges first). If the next edge does not induce a cycle among the current set of edges, then it is added to A. If it does, then this edge is passed over, and we consider the next edge in order. Note that as this algorithm runs, the edges of A will induce a forest on the vertices. As the algorithm continues, the trees of this forest are merged together, until we have a single tree containing all the vertices.

Observe that this strategy leads to a correct algorithm. Why? Consider the edge (u, v) that Kruskal's algorithm seeks to add next, and suppose that this edge does not induce a cycle in A. Let A' denote the tree of the forest A that contains vertex u. Consider the cut $(A', V - A')$. Every edge crossing the cut is not in A, and so this cut respects A, and (u, v) is the light edge across the cut (because any lighter edge would have been considered earlier by the algorithm). Thus, by the MST Lemma, (u, v) is safe.

The only tricky part of the algorithm is how to detect efficiently whether the addition of an edge will create a cycle in A. We could perform a DFS on subgraph induced by the edges of A, but this will take too much time. We want a fast test that tells us whether u and v are in the same tree of A.

This can be done by a data structure (which we have not studied) called the disjoint set Union-Find data structure. This data structure supports three operations:

Create-Set(u): Create a set containing a single item v.

Find-Set(u): Find the set that contains a given item u.

Union(u, v): Merge the set containing u and the set containing v into a common set.

You are not responsible for knowing how this data structure works (which is described in CLRS). You may use it as a "black-box". For our purposes it suffices to know that each of these operations can be performed in $O(\log n)$ time, on a set of size n. (The Union-Find data structure is quite interesting, because it can actually perform a sequence of n operations much faster than $O(n \log n)$ time. However we will not go into this here. $O(\log n)$ time is fast enough for its use in Kruskal's algorithm.)

In Kruskal's algorithm, the vertices of the graph will be the elements to be stored in the sets, and the sets will be vertices in each tree of A. The set A can be stored as a simple list of edges. The algorithm is shown below, and an example is shown in Fig. 34.

Kruskal's Algorithm

```
Kruskal(G=(V,E),w) {
    A = {}                                  // initially A is empty
    for each (u in V) Create_Set(u)         // create set for each vertex
    Sort E in increasing order by weight w
    for each ((u,v) from the sorted list) {
        if (Find_Set(u) != Find_Set(v)) {   // u and v in different trees
            Add (u,v) to A
            Union(u, v)
        }
    }
    return A
}
```

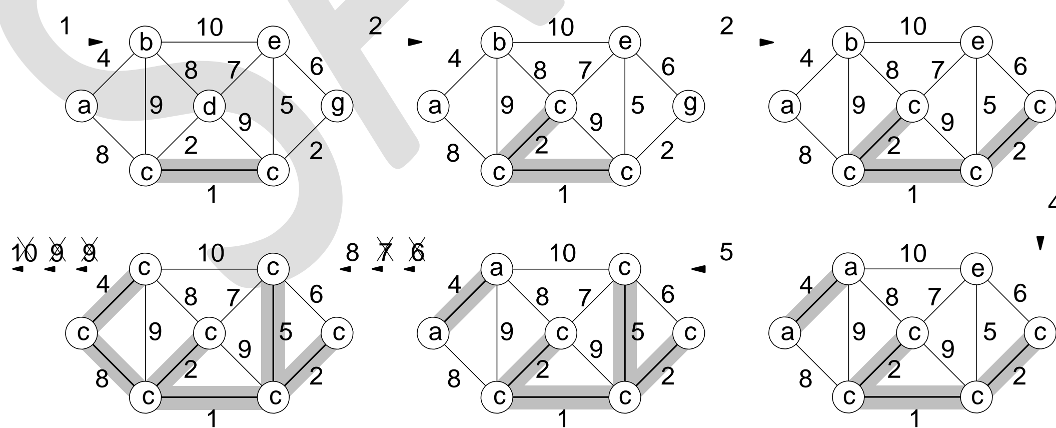


Fig. 34: Kruskal's Algorithm. Each vertex is labeled according to the set that contains it.

Analysis: How long does Kruskal's algorithm take? As usual, let V be the number of vertices and E be the number of edges. Since the graph is connected, we may assume that $E \geq V - 1$. Observe that it takes $\Theta(E \log E)$ time to

sort the edges. The for-loop is iterated E times, and each iteration involves a constant number of accesses to the Union-Find data structure on a collection of V items. Thus each access is $\Theta(V)$ time, for a total of $\Theta(E \log V)$. Thus the total running time is the sum of these, which is $\Theta((V + E) \log V)$. Since V is asymptotically no larger than E, we could write this more simply as $\Theta(E \log V)$.

Lecture 13: Prim's and Baruvka's Algorithms for MSTs

Read: Chapt 23 in CLRS. Baruvka's algorithm is not described in CLRS.

Prim's Algorithm: Prim's algorithm is another greedy algorithm for minimum spanning trees. It differs from Kruskal's algorithm only in how it selects the next *safe edge* to add at each step. Its running time is essentially the same as Kruskal's algorithm, $O((V + E) \log V)$. There are two reasons for studying Prim's algorithm. The first is to show that there is more than one way to solve a problem (an important lesson to learn in algorithm design), and the second is that Prim's algorithm looks very much like another greedy algorithm, called Dijkstra's algorithm, that we will study for a completely different problem, shortest paths. Thus, not only is Prim's a different way to solve the same MST problem, it is also the same way to solve a different problem. (Whatever that means!)

Different ways to grow a tree: Kruskal's algorithm worked by ordering the edges, and inserting them one by one into the spanning tree, taking care never to introduce a cycle. Intuitively Kruskal's works by merging or splicing two trees together, until all the vertices are in the same tree.

In contrast, Prim's algorithm builds the tree up by adding leaves one at a time to the current tree. We start with a root vertex r (it can be *any* vertex). At any time, the subset of edges A forms a single tree (in Kruskal's it formed a forest). We look to add a single vertex as a leaf to the tree. The process is illustrated in the following figure.

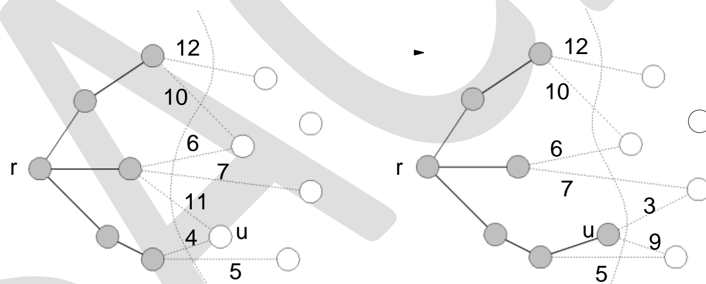


Fig. 35: Prim's Algorithm.

Observe that if we consider the set of vertices S currently part of the tree, and its complement $(V - S)$, we have a cut of the graph and the current set of tree edges A respects this cut. Which edge should we add next? The MST Lemma from the previous lecture tells us that it is safe to add the *light edge*. In the figure, this is the edge of weight 4 going to vertex u. Then u is added to the vertices of S, and the cut changes. Note that some edges that crossed the cut before are no longer crossing it, and others that were not crossing the cut are.

It is easy to see, that the key questions in the efficient implementation of Prim's algorithm is how to update the cut efficiently, and how to determine the light edge quickly. To do this, we will make use of a *priority queue* data structure. Recall that this is the data structure used in HeapSort. This is a data structure that stores a set of items, where each item is associated with a *key* value. The priority queue supports three operations.

insert(*u, key*)**:** Insert u with the key value *key* in Q.

extractMin()**:** Extract the item with the minimum key value in Q.

decreaseKey(u, ***new_key***)**:** Decrease the value of u's key value to *new_key*.

A priority queue can be implemented using the same heap data structure used in heapsort. All of the above operations can be performed in $O(\log n)$ time, where n is the number of items in the heap.

What do we store in the priority queue? At first you might think that we should store the edges that cross the cut, since this is what we are removing with each step of the algorithm. The problem is that when a vertex is moved from one side of the cut to the other, this results in a complicated sequence of updates.

There is a much more elegant solution, and this is what makes Prim's algorithm so nice. For each vertex in $u \in V - S$ (not part of the current spanning tree) we associate u with a key value $key[u]$, which is the weight of the lightest edge going from u to any vertex in S. We also store in $pred[u]$ the end vertex of this edge in S. If there is not edge from u to a vertex in $V - S$, then we set its key value to $+\infty$. We will also need to know which vertices are in S and which are not. We do this by coloring the vertices in S black.

Here is Prim's algorithm. The root vertex r can be any vertex in V.

Prim's Algorithm

```
Prim(G,w,r) {
    for each (u in V) {                    // initialization
        key[u] = +infinity;
        color[u] = white;
    }
    key[r] = 0;                            // start at root
    pred[r] = nil;
    Q = new PriQueue(V);                   // put vertices in Q
    while (Q.nonEmpty()) {                 // until all vertices in MST
        u = Q.extractMin();                // vertex with lightest edge
        for each (v in Adj[u]) {
            if ((color[v] == white) && (w(u,v) < key[v])) {
                key[v] = w(u,v);           // new lighter edge out of v
                Q.decreaseKey(v, key[v]);
                pred[v] = u;
            }
        }
        color[u] = black;
    }
    [The pred pointers define the MST as an inverted tree rooted at r]
}
```

The following figure illustrates Prim's algorithm. The arrows on edges indicate the predecessor pointers, and the numeric label in each vertex is the key value.

To analyze Prim's algorithm, we account for the time spent on each vertex as it is extracted from the priority queue. It takes $O(\log V)$ to extract this vertex from the queue. For each incident edge, we spend potentially $O(\log V)$ time decreasing the key of the neighboring vertex. Thus the time is $O(\log V + deg(u) \log V)$ time. The other steps of the update are constant time. So the overall running time is

$$\begin{aligned} T(V,E) &= \sum_{u \in V} (\log V + deg(u) \log V) = \sum_{u \in V} (1 + deg(u)) \log V \\ &= \log V \sum_{u \in V} (1 + deg(u)) = (\log V)(V + 2E) = \Theta((V + E) \log V). \end{aligned}$$

Since G is connected, V is asymptotically no greater than E, so this is $\Theta(E \log V)$. This is exactly the same as Kruskal's algorithm.

SACET

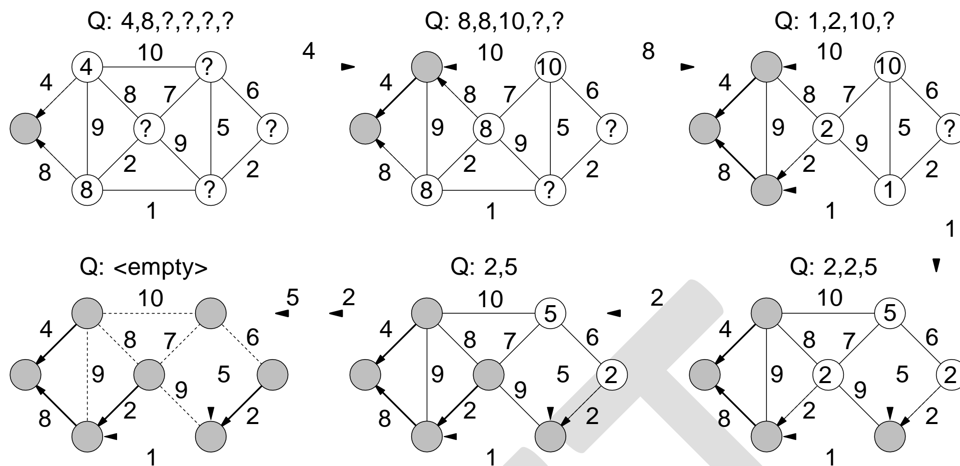


Fig. 36: Prim's Algorithm.

Baruvka's Algorithm: We have seen two ways (Kruskal's and Prim's algorithms) for solving the MST problem. So, it may seem like complete overkill to consider yet another algorithm. This one is called Baruvka's algorithm. It is actually the oldest of the three algorithms (invented in 1926, well before the first computers). The reason for studying this algorithm is that of the three algorithms, it is the easiest to implement on a parallel computer. Unlike Kruskal's and Prim's algorithms, which add edges one at a time, Baruvka's algorithm adds a whole set of edges all at once to the MST.

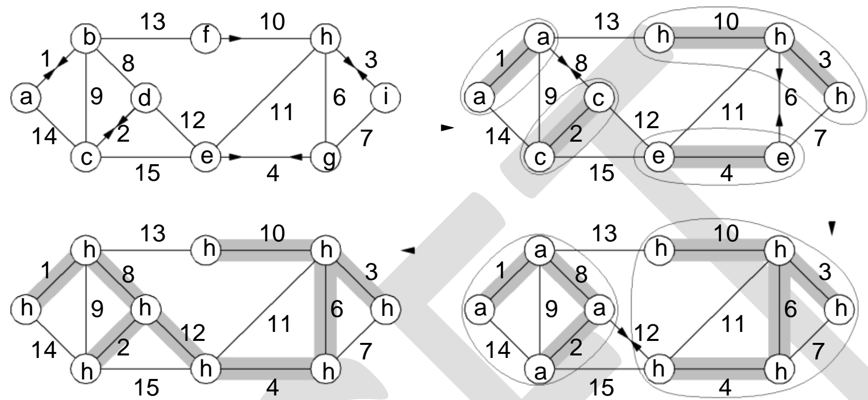
Baruvka's algorithm is similar to Kruskal's algorithm, in the sense that it works by maintaining a collection of disconnected trees. Let us call each subtree a *component*. Initially, each vertex is by itself in a one-vertex component. Recall that with each stage of Kruskal's algorithm, we add the lightest-weight edge that connects two different components together. To prove Kruskal's algorithm correct, we argued (from the MST Lemma) that the lightest such edge will be *safe* to add to the MST.

In fact, a closer inspection of the proof reveals that the cheapest edge leaving *any* component is always safe. This suggests a more parallel way to grow the MST. Each component determines the lightest edge that goes from inside the component to outside the component (we don't care where). We say that such an edge *leaves* the component. Note that two components might select the same edge by this process. By the above observation, all of these edges are safe, so we may add them all at once to the set A of edges in the MST. As a result, many components will be merged together into a single component. We then apply DFS to the edges of A, to identify the new components. This process is repeated until only one component remains. A fairly high-level description of Baruvka's algorithm is given below.

Baruvka's Algorithm

```
Baruvka(G=(V,E), w) {
    initialize each vertex to be its own component;
    A = {};                                 // A holds edges of the MST
    do {
        for (each component C) {
            find the lightest edge (u,v) with u in C and v not in C;
            add {u,v} to A (unless it is already there);
        }
        apply DFS to graph H=(V,A), to compute the new components;
    } while (there are 2 or more components);
    return A;                               // return final MST edges
```

There are a number of unspecified details in Baruvka's algorithm, which we will not spell out in detail, except to note that they can be solved in $\Theta(V + E)$ time through DFS. First, we may apply DFS, but only traversing the edges of A to compute the components. Each DFS tree will correspond to a separate component. We label each vertex with its component number as part of this process. With these labels it is easy to determine which edges go between components (since their endpoints have different labels). Then we can traverse each component again to determine the lightest edge that leaves the component. (In fact, with a little more cleverness, we can do all this without having to perform two separate DFS's.) The algorithm is illustrated in the figure below.



SACET

Fig. 37: Baruvka's Algorithm.

Analysis: How long does Baruvka's algorithm take? Observe that because each iteration involves doing a DFS, each iteration (of the outer do-while loop) can be performed in $\Theta(V + E)$ time. The question is how many iterations are required in general? We claim that there are never more than $O(\log n)$ iterations needed. To see why, let m denote the number of components at some stage. Each of the m components, will merge with at least one other component. Afterwards the number of remaining components could be a low as 1 (if they all merge together), but never higher than $m/2$ (if they merge in pairs). Thus, the number of components decreases by at least half with each iteration. Since we start with V components, this can happen at most $\lg V$ time, until only one component remains. Thus, the total running time is $\Theta((V + E) \log V)$ time. Again, since G is connected, V is asymptotically no larger than E, so we can write this more succinctly as $\Theta(E \log V)$. Thus all three algorithms have the same asymptotic running time.

SACET