

MA/CSSE 473

Day 34

**Optimal BST
Analysis**

Greedy Algorithms



Exam Algorithms

- Quite a few times I wrote something like "I can't follow this; please explain it to me and you may get more points."
- We'll do that now.



MA/CSSE 473 Day 34

- HW 13 due now
- HW 14 available
 - Includes a preview of a tough problem from HW 15
- Don't forget the Boyer-Moore demonstration animation, due Nov 20
- **Student Questions**
- Optimal BSTs
- Greedy algorithms



Recap: Optimal Binary Search Trees

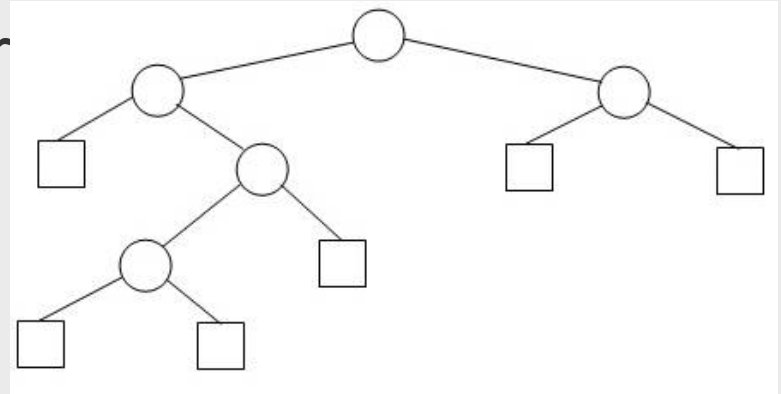
- Suppose we have n distinct data items x_1, x_2, \dots, x_n (arranged into increasing order) that we wish to arrange into a Binary Search Tree.
- This time the expected number of probes for an unsuccessful search depends on the shape of the tree and where the search ends up.
- Let y be the value we are searching for
- Let p_i be the probability that y is item x_i
- For $i = 1, \dots, n-1$ let q_i be the probability that $x_i < y < x_{i+1}$
- Similarly, let q_0 be the probability that $y < x_1$, and q_n the probability that $y > x_n$
- Note that
$$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$$

but we can (and will) also just use frequencies when finding the optimal tree (and divide by their sum to get probabilities).



Recap: Extended binary search tree

- Formally, an Extended Binary Search Tree (EBT) consists of
 - an external node, or
 - an (internal) root node and two EBTs T_L and T_R



- In diagram, Circles = internal nodes,
Squares = external nodes
- It's an alternative way of viewing a binary tree
- The external nodes stand for places where an unsuccessful search can end or where an element can be inserted
- An EBT with n internal nodes has $n + 1$ external nodes



What contributes to the expected number of probes?

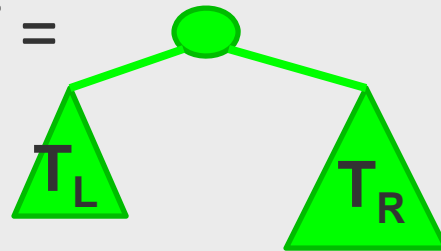
- Frequencies, depth of node
- For successful search, number of probes is one more than depth of the corresponding internal node
- For unsuccessful, number of probes is equal to depth of the corresponding external node



Weighted Path Length

$$\sum_{i=1}^n p_i [1 + \text{depth}(x_i)] + \sum_{i=0}^n q_i [\text{depth}(y_i)]$$

- If we divide this by $\sum p_i + \sum q_i$ we get the average search time.
- We can also define it recursively:
- $W(\square) = 0$. If $T =$



$W(T) = W(T_L) + W(T_R) + \sum p_i + \sum q_i$, where the summations are over all p_i and q_i for nodes in T

- It can be shown by induction that these two definitions are equivalent (good practice problem).



Example

- Frequencies of vowel occurrence in English
- : A, E, I, O, U
- p's: 32, 42, 26, 32, 12
- q's: 0, 34, 38, 58, 95, 21
- Draw a couple of trees (with E and I as roots), and see which is best. (sum of p's and q's is 390).



Strategy

- We want to minimize the weighted path length
- Once we have chosen the root, the left and right subtrees must themselves be optimal EBSTs
- We can build the tree from the bottom up, keeping track of previously-computed values



Intermediate Quantities

- Cost: Let C_{ij} (for $0 \leq i \leq j \leq n$) be the cost of an optimal tree (not necessarily unique) over the frequencies $q_i, p_{i+1}, p_{i+1}, \dots, p_j, q_j$. Then
- $C_{ii} = 0$, and
$$C_{ij} = \min_{i < k \leq j} (C_{i,k-1} + C_{kj}) + \sum_{t=i}^j q_t + \sum_{t=i+1}^j p_t$$
- This is true since the subtrees of an optimal tree must be optimal
- To simplify the computation, we define
- $W_{ii} = q_i$, and $W_{ij} = W_{i,j-1} + p_j + q_j$ for $i < j$.
- Note that $W_{ij} = q_i + p_{i+1} + \dots + p_j + q_j$, and so
- $C_{ii} = 0$, and
$$C_{ij} = W_{ij} + \min_{i < k \leq j} (C_{i,k-1} + C_{kj})$$
- Let R_{ij} be a value of k that minimizes $C_{i,k-1} + C_{kj}$ in the above formula



Code

```
# initialize the main diagonal
for i in range(n + 1):
    R[i][i] = i
    W[i][i] = q[i]
    drawSquare(i, i, W[i][i], C[i][i], R[i][i], win, inden)
# Now populate each of the n upper diagonals:
for d in range(1, n+1): # fill in this diagonal
    # The previous diagonals are already filled in.
    for i in range(n - d + 1):
        j = i + d; # on the dth diagonal, j - i = d
        opt = i + 1 # until we find a better one
        for k in range(i+2, j+1):
            if C[i][k-1]+C[k][j] < C[i][opt-1]+C[opt][j]:
                opt = k
        R[i][j] = opt
        W[i][j] = W[i][j-1] + p[j] + q[j]
        C[i][j] = C[i][opt-1] + C[opt][j] + W[i][j]
        drawSquare(i, j, W[i][j], C[i][j], R[i][j], win, i
```

Results

R00: 0	R01: 1	R02: 2	R03: 2	R04: 3	R05: 4
W00: 0	W01: 66	W02: 146	W03: 230	W04: 357	W05: 390
C00: 0	C01: 66	C02: 212	C03: 418	C04: 754	C05: 936
	R11: 1	R12: 2	R13: 3	R14: 3	R15: 4
	W11: 34	W12: 114	W13: 198	W14: 325	W15: 358
	C11: 0	C12: 114	C13: 312	C14: 624	C15: 798
		R22: 2	R23: 3	R24: 4	R25: 4
		W22: 38	W23: 122	W24: 249	W25: 282
		C22: 0	C23: 122	C24: 371	C25: 532
			R33: 3	R34: 4	R35: 4
			W33: 58	W34: 185	W35: 218
			C33: 0	C34: 185	C35: 346
				R44: 4	R45: 5
				W44: 95	W45: 128
				C44: 0	C45: 128
					R55: 5
					W55: 21
					C55: 0

**How to
construct the
optimal tree?**

**Analysis of the
algorithm?**

- Constructed by diagonals, from main diagonal upward
- What is the optimal tree?



Running time

- Most frequent statement is the comparison
if $C[i][k-1]+C[k][j] < C[i][opt-1]+C[opt][j]$:
- How many times
does it execute:

$$\sum_{d=1}^n \sum_{i=0}^{n-d} \sum_{k=i+2}^{i+d} 1$$

```
simplify(sum(sum(sum(1, k=i+2..i+d), i=0..n-d), d=1..n));
```

$$-\frac{1}{6}n + \frac{1}{6}n^3$$



Speeding it up

- `for k in range(i+2, j+1):`
 `if C[i][k-1]+C[k][j] < C[i][opt-1]+C[opt][j]:`
 `opt = k`
- If we change the `<` to `<=`, we always get the maximum possible `k`
- Then we can (but it takes some work to show it) replace `range(i+2, j+1)` by
 `range(R[i][j-1], R[i+1][j] + 1)`
- This makes the running time $\Theta(n^2)$



Greedy Algorithms

- Only for optimization problems
- Expand a partially-obtained solution until a complete solution is found.
- At each stage, the choice is
 - **feasible**: satisfies the problem's constraints
 - **locally optimal**: best choice among all available feasible choices at that step
 - **irrevocable**: Once made, the choice is not changes in subsequent steps of the algorithm



The Greedy Rule

- Whenever a choice is to be made, pick the one that seems best for the moment, without taking future choices into consideration.
- For example, a greedy Scrabble player will simply maximize the score for each turn, never saving any “good” letters for possible better plays later.
 - Doesn’t necessarily optimize score for entire game.



Greedy Chess

- Take a piece or pawn whenever you will not lose a piece or pawn (or will lose one of lesser value) on the next turn.
- Not a good strategy for this game either



Greedy Map Coloring

- On a planar map, choose a region and pick a color for that region
- Repeat until all regions are colored:
 - Choose a region R that is adjacent¹ to at least one colored region
 - Choose a color that is different than the colors of the regions that are adjacent to R
 - Use a color that has already been used if possible
 - Results in a valid map coloring, not necessarily minimum number of colors

¹ Two regions are adjacent if they have a common edge



Huffman's text Compression Algorithm

- You should have seen this in CSSE 230.
- Count the frequency of the various characters. Make one-character trees each of whose weight is its character's frequency.
- Repeat until only one tree is left:
 - Combine the two smallest-weight trees into a single tree with a new root whose weight is the sum of the weights of its two subtrees.
- Using this tree for encoding the message provides a code that is optimal among all single-character encodings of this message.



Coin-changing problem

- We have a collection of coins of various denominations.
- We want to be able to make change for A cents, using the smallest number of coins.
- Assume that we have as many coins of each denomination as we need.



Greedy coin-changing

$S = \emptyset$

Repeat until A is 0:

Choose (add to S) the largest-denomination coin that is less than A .

reduce A by the denomination of that coin.

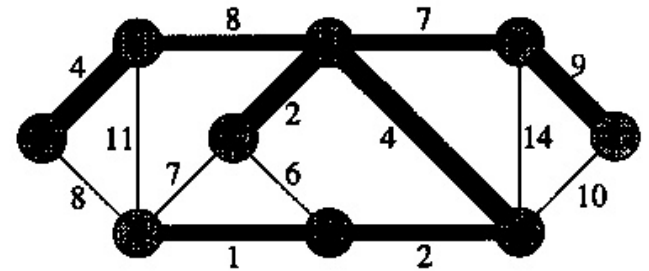
Return S

- Note that for some denomination sets, this algorithm always produces optimal solutions;
 - e.g., $\{10, 5, 1\}$
- for others it does not.
 - e.g., $\{10, 6, 1\}$ (try $A=12$)
- Later we may examine a non-greedy approach that always works.



Minimal Spanning Tree (MST)

- Suppose that we have a connected network G (a graph whose edges are labeled by numbers, which we call **weights**)
- We want to find a tree T that
 - spans the graph (i.e contains all nodes of G .)
 - minimizes (among all spanning trees) the sum of the weights of its edges.
- Is this MST unique?
- One approach: Generate all spanning trees and determine which is minimum
- Problems:
 - Number grows exponentially
 - Not easy to generate
 - Finding MST directly is simpler and faster



MST algorithms

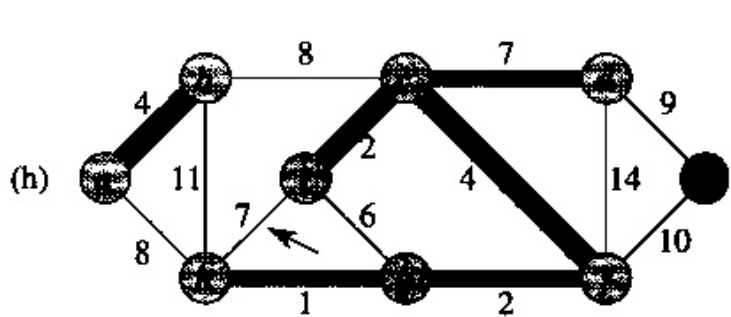
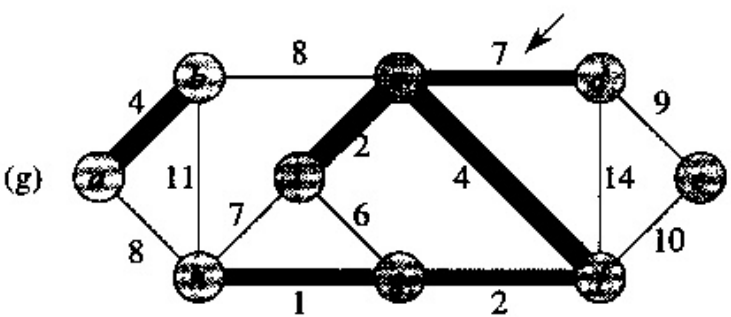
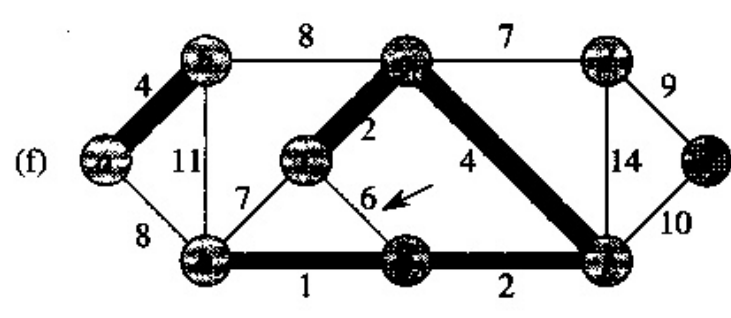
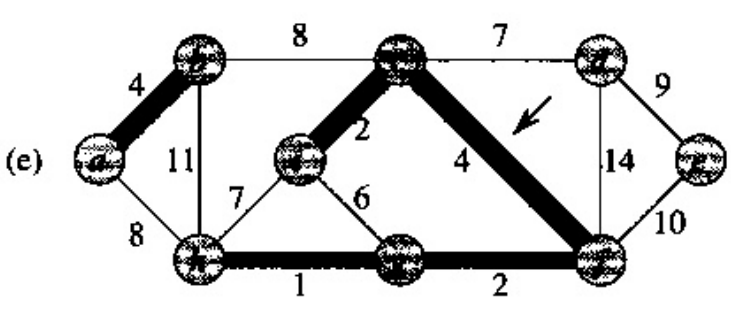
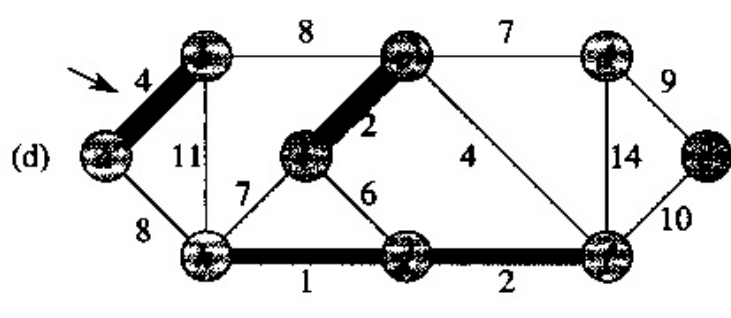
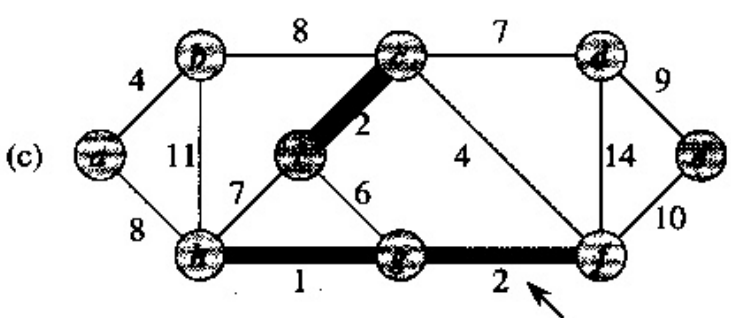
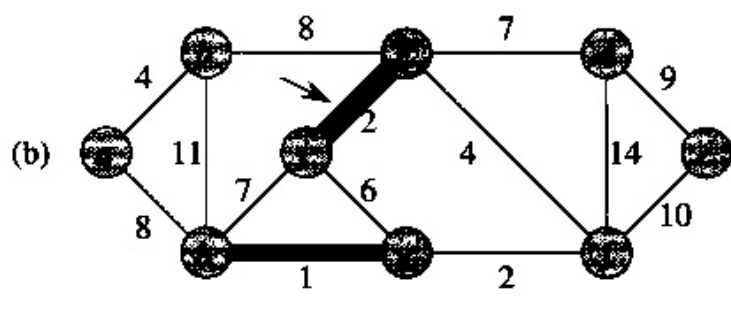
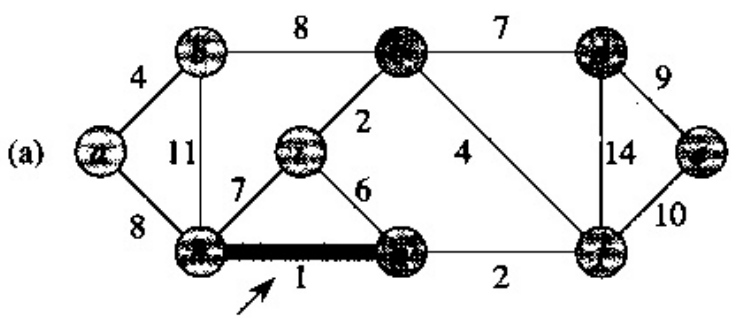
- Today we look at Prim's and Kruskal's algorithms
- Next time we think about proofs, implementation, efficiency

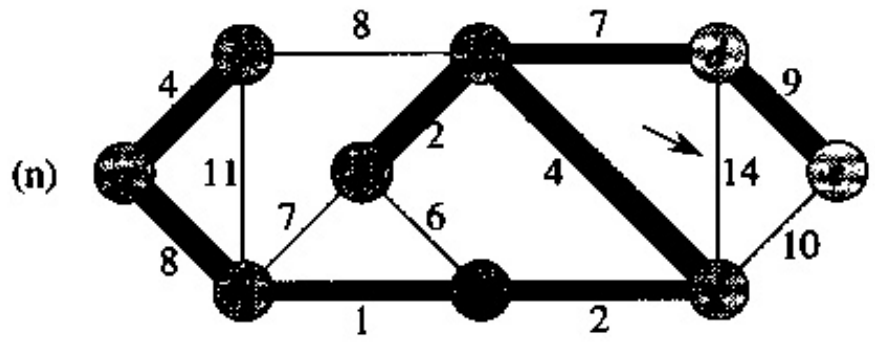
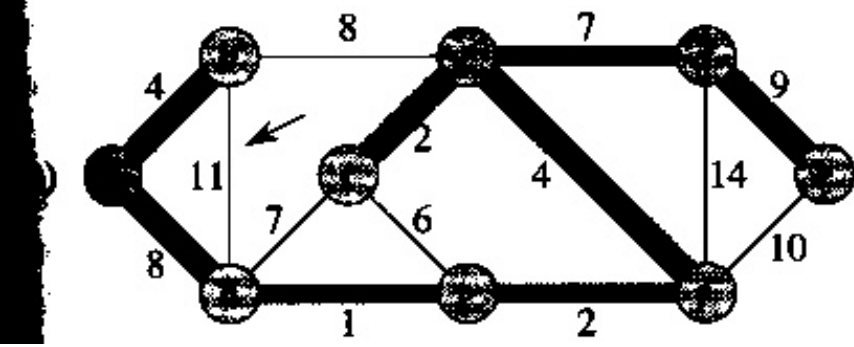
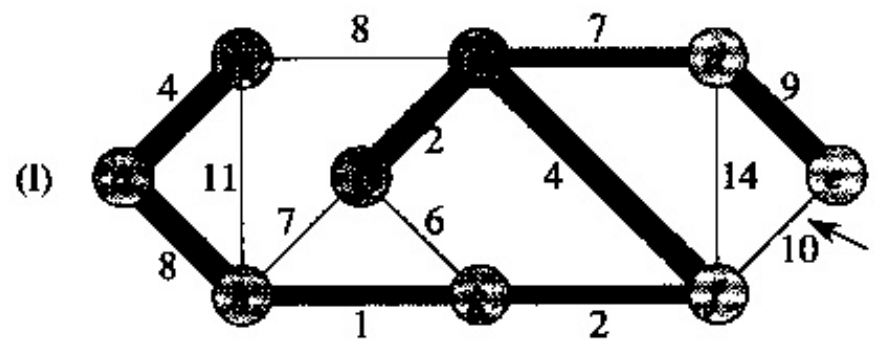
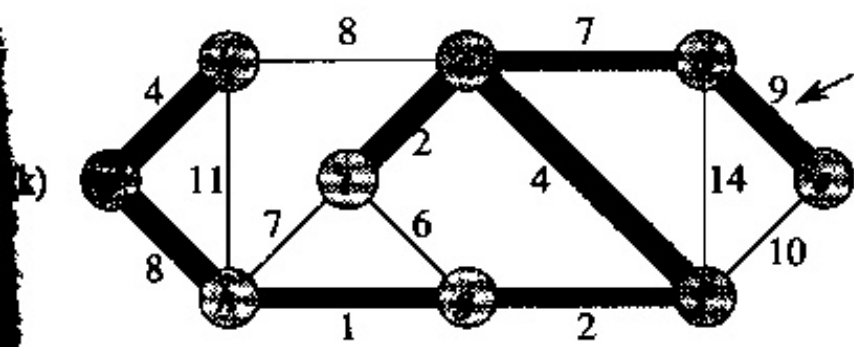
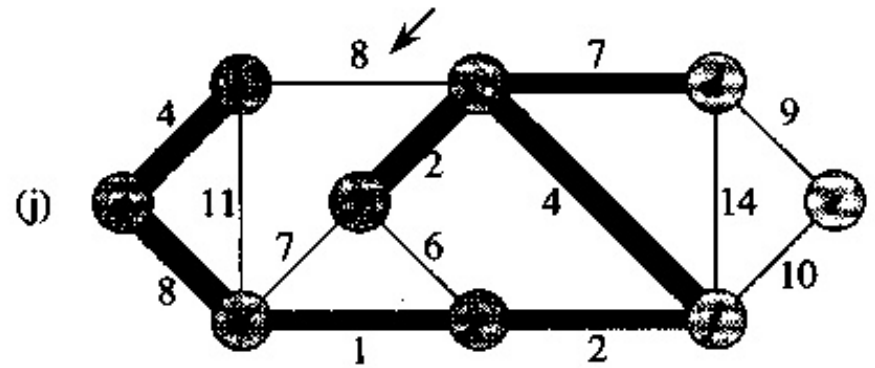
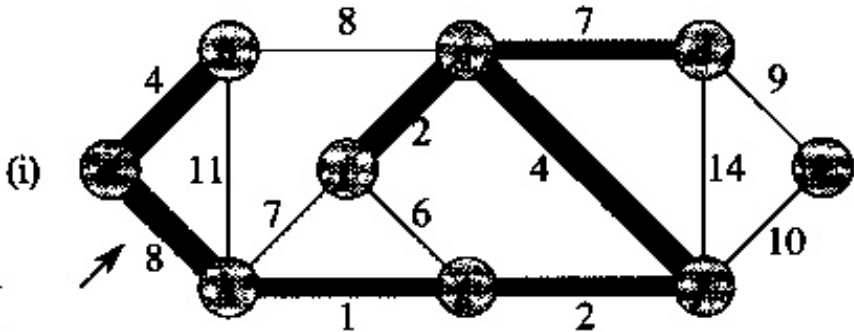


Kruskal's algorithm

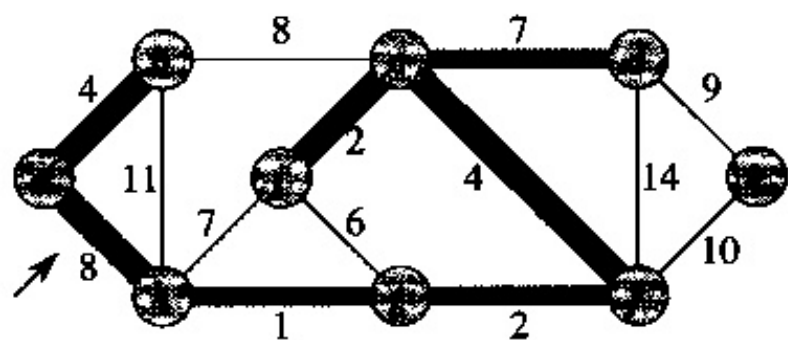
- To find a MST:
- Start with a graph containing all of G 's n vertices and none of its edges.
- for $i = 1$ to $n - 1$:
 - Among all of G 's edges that can be added without creating a cycle, add one with minimal weight.



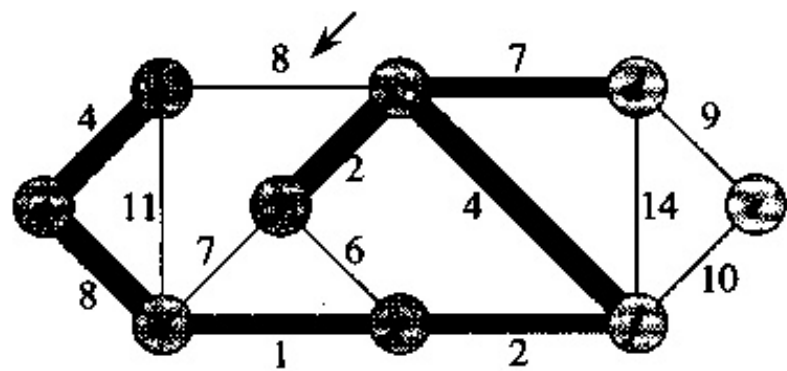




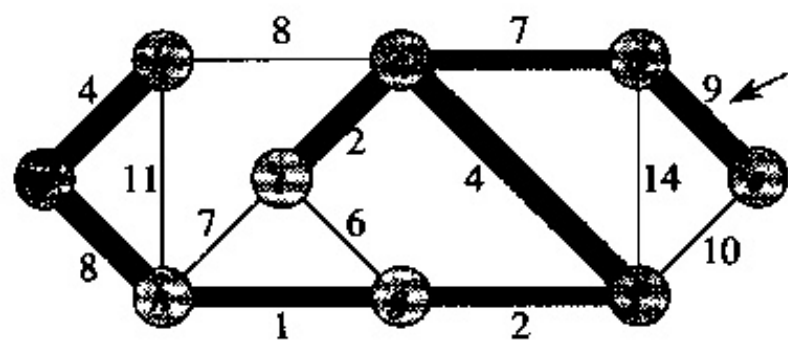
(i)



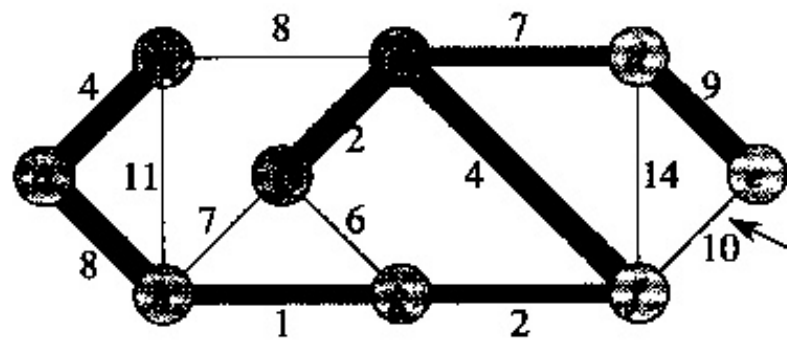
(j)



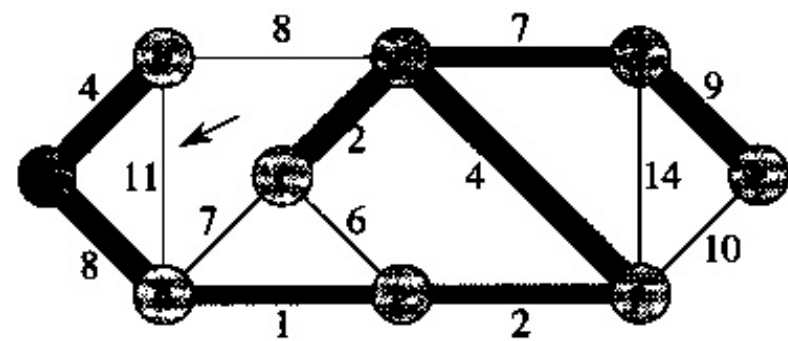
(k)



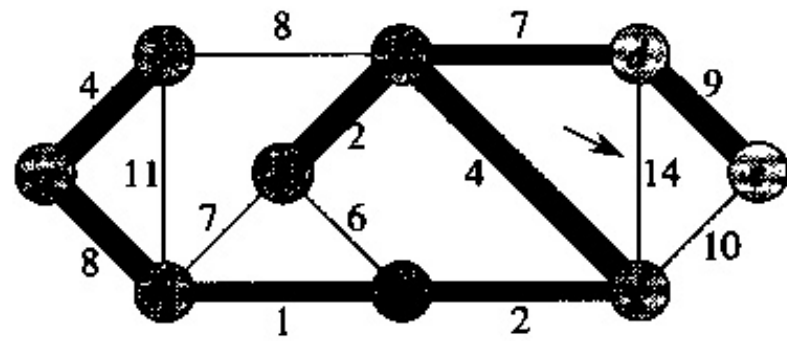
(l)



(m)



(n)



Prim's algorithm

- Start with T as a single vertex of G (which *is* a MST for a single-node graph).
- for $i = 1$ to $n - 1$:
 - Among all edges of G that connect a vertex in T to a vertex that is not yet in T , add to T a minimum-weight edge.



Example of Prim's algorithm

