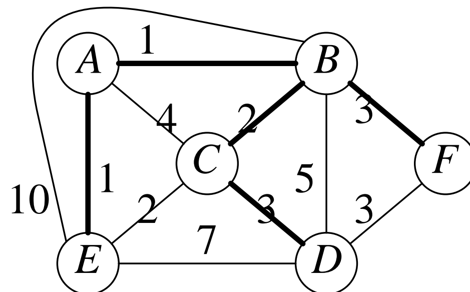
MINIMUM WEIGHT SPANNING TREE

Spanning Tree: Assume G is a weighted, connected graph.

- A tree $T = (V_T, E_T)$ contained in $G = (V, E)$, with $V_T = T$ and $E_T \subseteq E$.
- Weight $w(T) = \sum_{(x, y) \in T} w(x, y)$.

Min. Weight Spanning Tree (MST) T_0**:**

- $w(T_0) = \min \{w(T)$: T is a spanning tree of $G\}$.



An weighted graph G and an MST whose edges are shown in bold.

Questions:

•? How many other MST's are there in the above G?

A Brute-Force Algorithm:

1. Find all spanning trees and their costs.
2. Choose one with the minimum weight.

Complexity: Can be exponential in $N = |V|$.

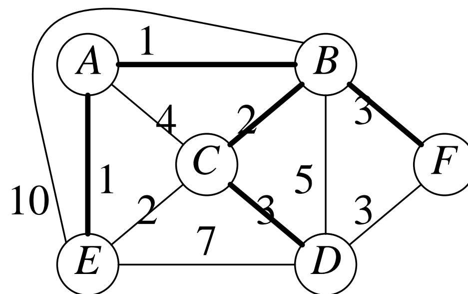
- The number of spanning trees in a complete graph with N nodes is N^{N-2}. (Finding them systematically without repetitions is non-trivial.)
- Computing $w(T)$ for each spanning tree T takes $O(N)$.

AN IMPORTANT PROPERTY OF MST

Structure Theorem for MST**.**

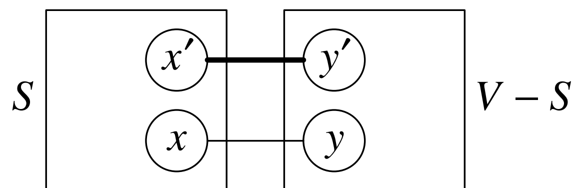
- Let $E(S, V - S)$ = The set of edges of G which connect a node in $S \subset V$ and a node in $V - S$.
- If (x, y) is a minimum-weight edge in $E(S, V - S)$, then there is an MST which contains (x, y).

Question: For $S = \{A, B, E\}$, what are some (x, y) below?



Can we get an MST containing (x, y) in each case?

Proof: Let T be an MST and $(x, y) \notin T$, and let C be the cycle in $T + (x, y)$ containing (x, y). Clearly, C contains at least one other edge (x', y') connecting S and $V - S$.

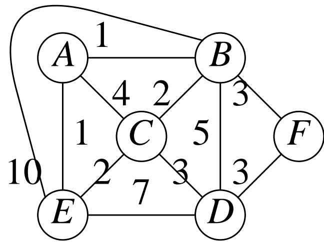


Since $w(x', y') \geq w(x, y)$ and $T' = T + (x, y) - (x', y')$ is a spanning tree and $w(T) \leq w(T') = w(T) + w(x', y') - w(x', y')$ it follows that $w(x', y') = w(x, y)$ and $w(T') = w(T)$ and T satisfies the theorem.

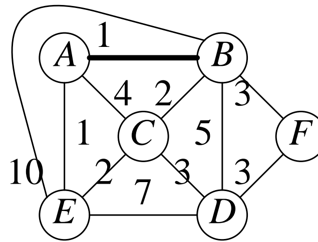
APPLICATION OF MST-STRUCTURE

A Method for Constructing an MST:

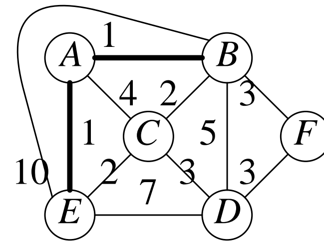
- Start with an arbitrary node x_0 and let $S = \{x_0\}$ and $T_E = \varnothing$.
- Successively add an edge to T based on the Structure-Theorem until T becomes a spanning tree.



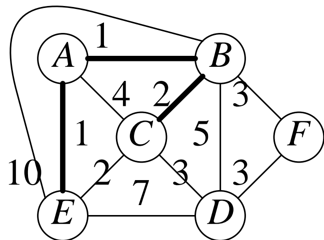
(i) $S = \{A\}$



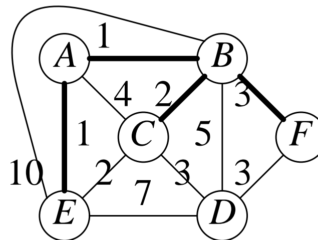
(ii) $S = \{A, B\}$; edge added (A, B)



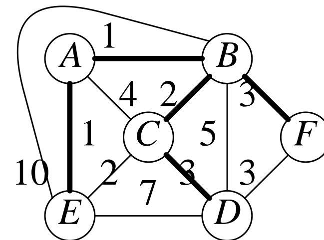
(iii) $S = \{A, B, E\}$; edge added (A, E)



(iv) $S = \{A, B, E, C\}$; edged added (B, C).



(v) $S = \{A, B, E, C, F\}$; edged added (B, F).



(vi) Edged added (C, D).

S (and edge added to T)	Edges between S and $V - S$ (new edges are underlined)	Reduced edge-set (one least weight edge at nodes in $V - S$)
$\{A\}$ (AB)	$\underline{AB, AC, AE}$	$\underline{AB, AC, AE}$ (none at D, F)
$\{A, B\}$ (AE)	$AC, AE, \underline{BC, BD, BE, BF}$	$AE, \underline{BC, BD, BF}$
$\{A, B, E\}$ (BC)	$AC, BC, BD, BF, \underline{EC, ED}$	BC, BD, BF
$\{A, B, E, C\}$ (BF)	$BD, BF, ED, \underline{CD}$	$\underline{CD}, BF$
$\{A, B, E, C, F\}$ (CD)	$BD, ED, CD, \underline{FD}$	BF

Question: For $|S| = k$, then how large can $E(S, V - S)$ be?

PRIM'S ALGORITHM

Notations:

- For $y \in V - S$, bestWeight(y) = min $\{w(x, y): x \in S\}$.
- best(y) = $x \in S$, where $w(x, y)$ = bestWeight(y).

Algorithm PRIM:

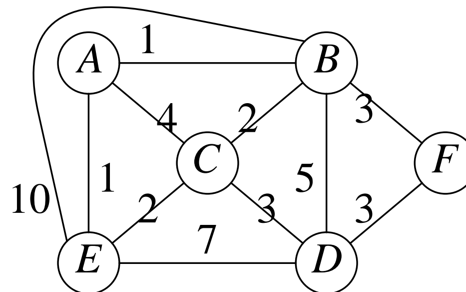
1. [Initialization.]
   (a) Choose a node x, let $S = \{x\}$ = the set of current nodes in T, and $E_T = \emptyset$, the set of edges in T.
   (b) For each $y \notin S$, let best(y) = x and bestWeight(y) = $w(x, y)$ if $y \in$ adjList(x) and otherwise let best(y) = y and bestWeight(y) = ∞ (a large number).
2. [Add a node and edge to T.] Repeat (a)-(b) $N - 1$ times:
   (a) Choose $x \notin S$ such that bestWeight(x) = min{bestWeight(y): $y \notin S$}; add x to S and (best(x), x) to E_T.
   (b) For each $y \notin S$, do the following:
   if $w(x, y) <$ bestWeight(y), then let best(y) = x and bestWeight(y) = $w(x, y)$.

Note: For efficient execution of step 2(c), which processes each edge of G once, represent G by an adjacency matrix, with the entries $w(x, y)$.

Complexity: $O(N^2)$.

- All iterations of step 2(a): $O((N - 1) + \cdots + 2 + 1) = O(N^2)$.
- All iterations of step 2(b): $O((N - 2) + \cdots + 2 + 1) = O(N^2)$.

ILLUSTRATION OF PRIM'S ALGORITHM



("−" means not relevant.)

Node	best(y) and bestWeight(y)						Edge added
x	A	B	C	D	E	F	to T
A	−	A, 1	A, 4	D, ∞	A, 1	F, ∞	
B	−	−	B, 2	B, 5	A, 1	B, 3	(A, B)
E	−	−	B, 2	B, 5	−	B, 3	(A, E)
C	−	−	−	C, 3	−	B, 3	(B, C)
D	−	−	−	−	−	B, 3	(C, D)
F	−	−	−	−	−	−	(B, F)

Question:

•? Show the execution of Prim's algorithm for the start node D.

• Show the execution of Prim's algorithm for the above graph with all weight w_i replaced by $-w_i$; assume the start node is A.

• Give the complexity of PRIM's algorithm if we use a brute force implementation of step (2) as indicated in the second column of the table on page 6.3, without using best(y) and bestWeight(y).

•? Let T_1 and T_2 be two MST's for a graph G and $n(T_i, w)$ = #(edges in T_i with weight w). Prove that $n(T_1, w) = n(T_2, w)$ for all w. (You can verify the equality for G on page 6.1.)

•? What is common to all spanning trees of a graph?

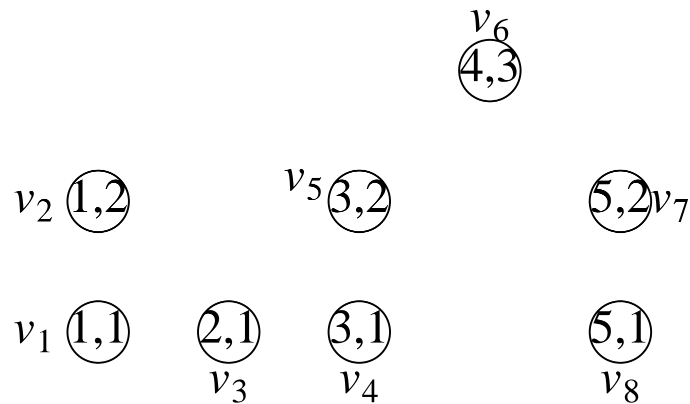
•? What is a common additional property of all MSTs of a weighted graph?

EXERCISE

1. Let $T = (V, E)$ be a tree with its edge-weights $w(v_i, v_j) > 0$. For each node-pair v_i and v_j, let $c(v_i, v_j) = |\pi(v_i, v_j)|$, where $\pi(v_i, v_j)$ is the unique $v_i v_j$-path in T. Consider the complete digraph G_T on the vertices V with $c(v_i, v_j) = c(v_j, v_i)$. How can you determine T from G_T?

   Illustrate your answer by showing G_T when we take T to be the MST on page 2.

2. Consider a set of points $V = \{v_i(x_i, y_i), 1 \le i \le N\}$ in the plane, where (x_i, y_i) are the coordinates of the point v_i. Shown below is an example for $N = 8$. For each $r > 0$, define the set of edges $E_r = \{(v_i, v_j): d(v_i, v_j) \le r\}$ among the nodes v_i. We want to find the smallest $r = r_c$ such that the edges E_r form a connected graph on the nodes V.



   (a) How are the sets E_r and $E_{r'}$ related (for an arbitrary V) for $r < r'$? Explain your answer for the above nodes.

   (b) Find $r = r_c$ for the example nodes V above; show E_r on the points V. Find the largest $r' < r_c$ such that $|E_{r'}| < |E_r|$ and show $E_{r'}$.

   (c) Give an algorithm for finding r_c for an arbitrary V. Explain the algorithm using the example set of nodes.

USE OF INPUT/OUTPUT STRUCTURES IN PRIM'S ALGORITHM

Use Of Input Structure:

- A change in the weights of edges may change the reduced edge-set considered in each iteration of Step 2 and the successive edges (x, y) selected in Step 2(a), but it still looks at every edge once in updating $best(z)$ and $c(z)$ for $z \in V - V_T$ in the course of the algorithm and no short-cuts is made.

  Thus, it *does not* make use of any input-structure. A change in the edges of the graph can be thought of as a change in the weights (why?).

Use Of Output Structure:

- Cycles are avoided by adding each time an edge connecting a node in V_T to a node not in V_T.

  Thus, the output-structure is *used*; no attempt is made to determine which part of the final tree is to be created first.

Use Of Input-Output Relationship:

- The essential part of the algorithm (choosing a smallest cost edge between V_T and $V - V_T$) is based on the input-output relationship $T \subseteq G$.

  Thus (as almost in all algorithms), the input-output relationship is used here.

GREEDY NATURE OF PRIM'S ALGORITHM

- At each iteration, we choose a *locally* optimal ("greedy" choice) is made of an edge e (which involves some computation) from the edges between V_T and $V - V_T$.
- Once a choice is made, it is never revised and is a part of the final solution (i.e., no backtracking).

Question:

(1) In what way, Dijkstra's algorithm for shortest x-y path is not "greedy"? (What local decisions are made at any stage and which of them may be revised subsequently?)

In what sense, Dijkstra's algorithm for shortest-path from x to all nodes (reachable from x) is greedy?

(2) In what way, Floyd's algorithm is not "greedy"?

(3) In some sense, insertion sort is greedy and in some sense it is not? Explain (use an actual example array of numbers, if needed.).

(4) Is the selection-sort greedy? How about the Heap-sorting?

KRUSKAL'S ALGORITHM FOR MINIMUM-WEIGHT SPANNING TREE

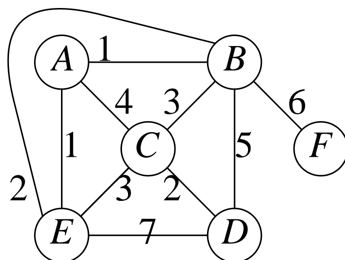
Idea: (1) Successively add edges of *smallest* weights, without creating a cycle.

(2) Each intermediate stage consist of a *forest* on the nodes of G. (This is a slight disadvantage compared to Prim's algorithm.)

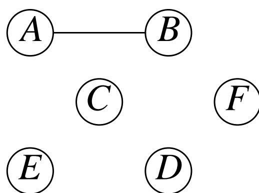
Advantage Over Prim's algorithm:

Gives a more efficient implementation, using the *disjoint set-union* algorithm, if G has $O(N^{\alpha})$, $\alpha < 2$, edges.

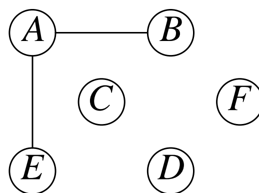
Example.



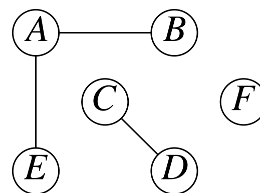
Order edges in non-decreasing weights:
$(A, B) = 1$, $(A, E) = 1$, $(B, E) = 2$, $(C, D) = 2$, $(B, C) = 3$,
$(C, E) = 3$, $(A, C) = 4$, $(B, D) = 5$, $(B, F) = 6$, $(D, E) = 7$.



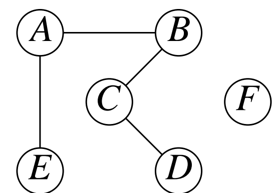
(A, B) added.



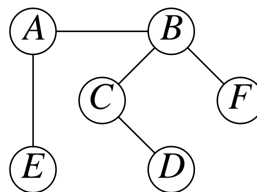
(A, E) added.



(B, E) not added;
(C, D) added.



(B, C) added.



(C, E), (A, C), (B, D)
not added; (B, F) added.

KRUSKAL'S ALGORITHM

Algorithm KRUSKAL:

Input: A weighted connected graph $G = (V, E)$.
Output: The edges E_T of a min-weight spanning tree of G.

1. [Initialize.] Sort the edges in the non-decreasing order of weights. Initialize a set $\{x\}$ for each node x in G, and initialize $E_T = \emptyset$.
2. For (each successive edge (x, y)) do the following until $|E_T| = |V| - 1$:

   If (x and y belong to different sets), then add (x, y) to E_T and merge those two sets into a single set (throwing away the old ones).

EXERCISE

1. If we use an ordinary list representation of the various sets and each of the first $|V| - 1$ edges are added to E_T, then what is the complexity of processing the ith edge?
2. Give an example graph to illustrate the following extreme scenario: we add the edges e_1, e_2, and do not add e_3, then we add e_4 and not add the edges $\{e_5, e_6, e_7\}$, then add e_8 and not add $\{e_9, e_{10}, e_{11}, e_{12}\}$, etc.
3. Compare the use of input/output structure in Kruskal's algorithm with those in Prim's algorithm.

EFFICIENT IMPLEMENTATION OFi KRUSKAL'S ALGORITHM

Set-operations used in Kruskal's algorithm:

(1) *Find* the set containing a given element z ($z = x$ and $z = y$ for an edge (x, y)).

(2) *Merge* the sets containing x and y if they are different (and hence disjoint).

Data-structure: Each set is represented by a tree (a node may have more than 2 children).

(1) If $r(x)$ = root of the tree containing x, then x and y are in the same set if and only if $r(x) = r(y)$.

(2) To merge the sets containing x and y, make the tree with the root $r(x)$ a child of $r(y)$ if the tree with root $r(x)$ has smaller size than the tree with root $r(y)$. (This keeps the height of the trees small and hence makes the determination of $r(x)$ effi - cient.)

Complexity: $O(|E| \log N) < (N^2)$ if $|E| = O(N^\alpha)$ and $\alpha < 2$.

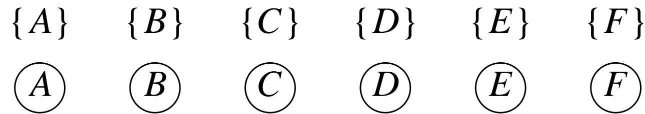
(1) Sorting edges take $O(|E| \log |E|) = O(|E| \log N)$ time.

(2) Height of a tree with k nodes can be shown to be $O(\log k)$.

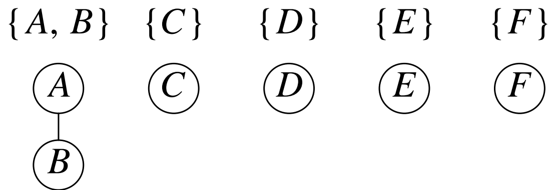
(3) For each edge (x, y), fi nding $r(x)$ and $r(y)$ takes $O(\log N)$ time by following parent-links. Testing "$r(x) = r(y)$" and then merging the trees, if necessary, take $O(1)$ time.

(4) We update the size of the tree when two trees are merged.

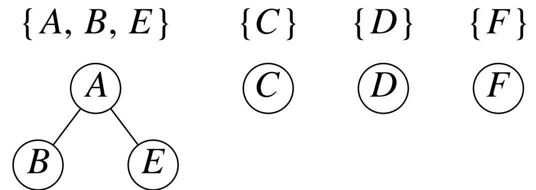
Example: For the graph shown earlier:



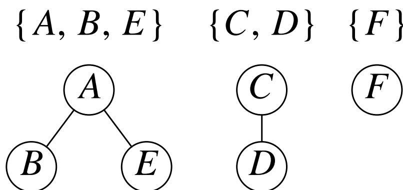
(i) Initial sets having one node each and their associated trees.



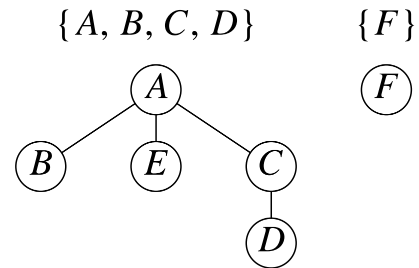
(ii) After processing edge (A, B).



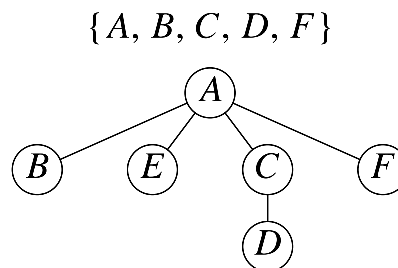
(iii) After processing edge (A, E).



(iv) After processing edge (C, D).



(v) After processing edge (B, C).



(vi) After processing edge (B, F).

EXERCISE

1. Show all possible rooted trees with N = 3, 4, 5, and 6 nodes. (Keep the subtrees of a node with larger size on the left. This means there are only 3 such trees for N = 4.) For each tree, show all possible sequence of merge-operations, if any, that give rise to that tree. (Label the edges as 1, 2, ⋯, $N - 1$ to indicate the order in which the subtrees were merged.)

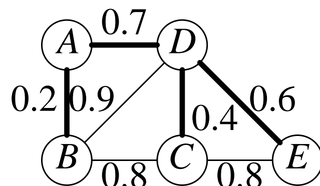
RECURSIVE ALGORITHM FOR MIN-SPANNING TREE

Reduction: Consider $G - x$ for some node x.

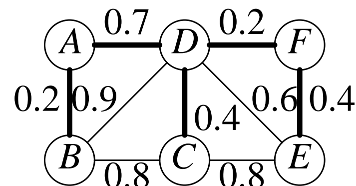
Algorithm:

1. Choose a node x.
2. Construct a min-spanning tree on each connected component C_i of $G - x$.
3. For (each C_i) do the following:
   (a) Add the smallest weight edge from x to C_i.
   (b) For (each of the remaining edges e_i from x to C_i), eliminate a largest weight edge in the cycle in $T_i + e_i$ and call the new tree T_i.

Problem: This may require throwing away some edges already selected. In particular, this is not a "greedy" solution.



(i) MST on $G - x$ for $x = F$.



(ii) MST on G; (F, D) and (F, E) are added, and (D, E) is deleted.

Possible solution: Choose x in such a way that no edge needs to be thrown away once selected.

Question: (1) Does there always exist such an x?
(2) Is there an efficient way of finding such an x?

EXERCISE

1. Examine Prim's algorithm carefully to see if we could use information generated during the construction of an MST T' on $G - x$ (assume for the moment that $G - x$ is connected, if that helps you) that can be used to quickly determine what edges from x should be deleted from T' and which edges should be added to T' to get an MST for G.
2. Does the above method give a better complexity than $O(N^2)$?

ANOTHER RECURSIVE APPROACH FOR THE MIN-WEIGHT SPANNING TREE PROBLEM

Reduction: Delete an edge e and consider $G - e$.

Algorithm:

1. Choose a *smallest* weight edge e in $G = (V, E)$.
2. Find an MST for each connected component of $G' = G - e$. (#Components ≤ 2.)
3. If (G' is disconnected) then add e to the MST's T_1 and T_2 for the two components of G' else do the following;

   Find a largest weight edge e' in the cycle obtained by adding e to the MST T in G', and replace e' by e.

Notes:

- If we replace "smallest" in step (1) by "largest", then we do not need "else" part in step (3). This does not mean that the modified form is going to be necessarily more efficient (why?).
- Finding connected components of G' can be done in linear time (in #edges) by the depth-first search.
- Detection of cycle in step (3) can be done by a depth-first search of MST T, starting at an end point of e; this takes $O(|V|)$ time.

Comment: This is not more efficient than the non-recursive algorithms of Prim and Kruskal.

Question: What is the worst case complexity of first approach, using e = the largest weight edge.

GENERATING ALL SPANNING TREES IN A CONNECTED GRAPH

Given: A connected graph $G = (V, E)$ with N nodes.
Find: All spanning trees in G.

Key Ideas:

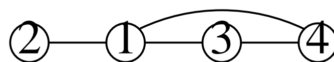
(1) Build each spanning tree T in a depth-first fashion (using, say, node 1 as the start node); in this way, we can make sure that each T is generated exactly once.

(2) The successive extensions in building T can be considered a path from the root node in a search-tree for building all spanning trees. The root node of the search-tree corresponds to the partial tree consisting of just node 1. Each terminal node of the search-tree represents either a spanning tree or a partial tree that cannot be completed to a span. tree.

(3) Given a partial tree T and the possible choices $E = \{(x_j^T, y_j^T)$: $j \geq 0\}$ for the next edge in extending T, the selection of (x_k^T, y_k^T) from E is equivalent to pretending that (x_i^T, y_i^T), $i < k$ are not present in the graph G.

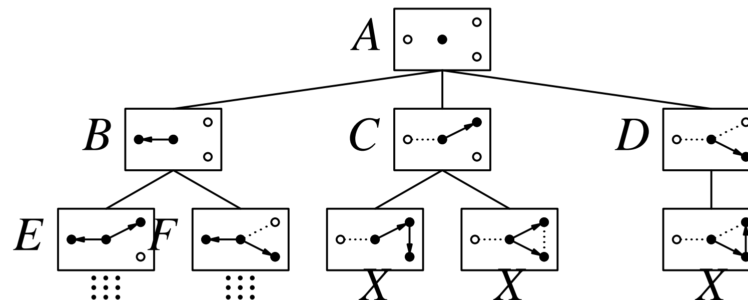
(4) If (x_k, y_k) is the kth edge in T, then its selection depends only on the knowledge of the end point y_{k-1} of the previous edge (x_{k-1}, y_{k-1}) added to T and the current nodes in T; x_k is either y_{k-1} or an ancestor of y_{k-1} in T.

Question: What are the spanning trees of the graph G below?



Search Tree: Shown below is part of the search-tree.

- At each node of the search-tree, the solid circles shows the nodes in the partial tree T; the edges of T are shown directed from parents to children.
- The dotted lines show the edges of G that are unavailable for building T. The number of dotted edges increase from left to right among the brothers; A node and its leftmost child in the search-tree have the same dotted lines.
- A terminal node of the search-tree which is not a spanning tree is marked "X".
- The node E in the search tree will give rise to two spanning trees, and the node F to one.



Part of the search-tree for spanning trees of the graph G shown above.

EXERCISE

1. If x and y are two brother nodes in the search-tree and all terminal nodes in the search-tree which are descendents of x are marked "X", then argue that the same is true for the terminal nodes that are descendents of y. Verify the statement for the above search-tree.

PSEUDOCODE FOR GENERATING ALL SPANNING TREES IN A CONNECTED GRAPH

Key Features:

- Avoids building the search-tree itself to reduce memory-use. The current chain of recursive calls correspond to the path in the search-tree from its root node to the current node.
- Keeps only one partial tree T at any point in the execution of the algorithm. As we go down in the chain of recursive calls, T is extended by adding an edge (see step 4(a) below), and a back-tracking in the search-tree (i.e., a return of a call) corresponds to the removal of the latest edge added to T (see step 4(b) below).
- If j is the latest node added to T, then the set E in step (2) below may contain zero or more edges of G adjacent to j or to an ancestor of j. To reduce the search of adjList(i) for G in determining E, one can use an array whereIs[i,k] = a pointer to the item for node k in adjList(i). If i is an ancestor of j in T and k is the latest child of i in T, then adjList(i) need to be search only after the item whereIs[i,k].

Algorithm EXPAND(T): //T = empty tree in initial call.

Input:	A connected graph G, in the adjacency list form.
Output:	Computes all spanning trees of G by successively extending the current partial tree T.

1. If $T = \emptyset$, then initialize T consisting a of a single node (say, node 1), mark this node as "visited", initialize numMarkedNodes = 1, let lastNodeAdded = 1, and call Expand(T).
2. Determine the set of edges $E = \{(x_k, y_k)$: T can be extended in a depth-first fashion by the addition of (x_k, y_k), pretending that the previous edges (x_i, y_i), $i < k$ are not available if $k > 1\}$, (Use the

current T and lastNodeAdded = the last node added to T for this purpose.)

3. If ($E = \emptyset$), then do the following:

   If (numMarkedNodes = numNodesInGraph), then output the spanning tree T (else T is not a spanning tree and cannot be extended further).

4. Otherwise, for each $(x_k, y_k) \in E$ do the following:

   (a) Let $T = T + (x_k, y_k)$, mark the node y_k, let numMarkedNodes = numMarkedNodes + 1, let lastNodeAdded = y_k, and call Expand(T).

   (b) Unmark the node y_k, delete the edge (x_k, y_k) from T, and reset numMarkedNodes = numMarkedNodes − 1 for the next iteration of (a)-(b). (No need to reset lastNodeAdded.)

EXERCISE

1. Modify the pseudocode to improve the efficiency by avoiding the generation of recursive calls to Expand corresponding to brother nodes in the search-tree when the call at the previous brother did not generate any spanning tree. For example, this would avoid the call corresponding to the search-tree node D in the Fig. on previous page.

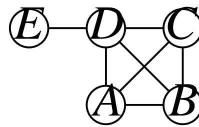
2. Modify the pseudocode to improve the efficiency even further by avoiding the generation of a recursive call which will not give rise to a spanning tree because the most recent edge in G that is being regarded as "not present" happens to be a cut-edge (when considered together with other edges of G that are currently being considered being not present). For example, this would avoid the calls corresponding to both the search-tree nodes C and D in the Fig. on previous page.

Compare the each of the following method with the method given above in terms of where it encompasses major efficiency loss.

(1) Consider all $(N-1)$-subsets E_j of the edges, and select those which do not contain a cycle. (Note that determining a cycle within E_j requires non-trivial computation.)

(2) Consider all spanning-forest of G and arrange them in the form of a tree T such that: (a) each terminal node of T gives a spanning-tree of G, and (b) if $F(x)$ = the spanning-forest of G for node x in T and x = parent(y), then $F(y) = F(x) + (u, v)$, where $(u, v) \in E$. Finally, create T in a depth-first fashion and print the terminal nodes.

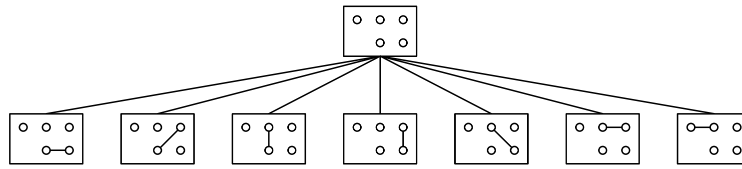
Question: How can use information from previous failed E_j's for the other E_j's in approach (1)? Explain your ideas using the graph G below.



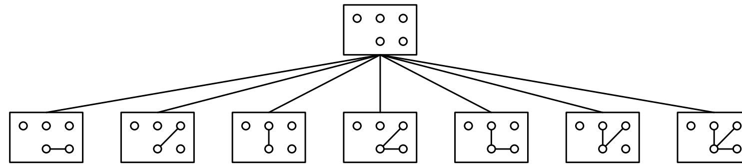
Ideas in reducing computation in Approach (2):

(1) Label the edges E as $e_1, e_2, \cdots$ in some order so that if e_i is the last edge added in obtaining $F(x)$ at x, then for a child y of x we can add e_j only for $j > i$. This assures that no two forests will be the same and hence each spanning tree will be created only once.

(2) Keep a list $L(x) = \{e_k: e_k$ forms a cycle with $F(x)\}$, and update $L(x)$ as you go down T. This saves consideration of certain edges for cycle formation at descendents of x.



The children of the root node for the ordering:
$(A, B) < (A, C) < (A, D) < (B, C) < (B, D) < \cdots$.



An alternative branching from the root of the search-tree,
where a child y may have two or more edges added to obtain $F(y)$;
the two methods may give different number of children.

EXERCISE

1. Show part of the tree T for the above G to explain the use of L.
2. What are the problems in the following approach. Let the root of T be G itself. Let $G(x)$ denote the graph at a node $x \in T$. We select a cycle C of $G(x)$, and then for each edge $e \in C$ we create one child of x corresponding to $G(x) - e$.