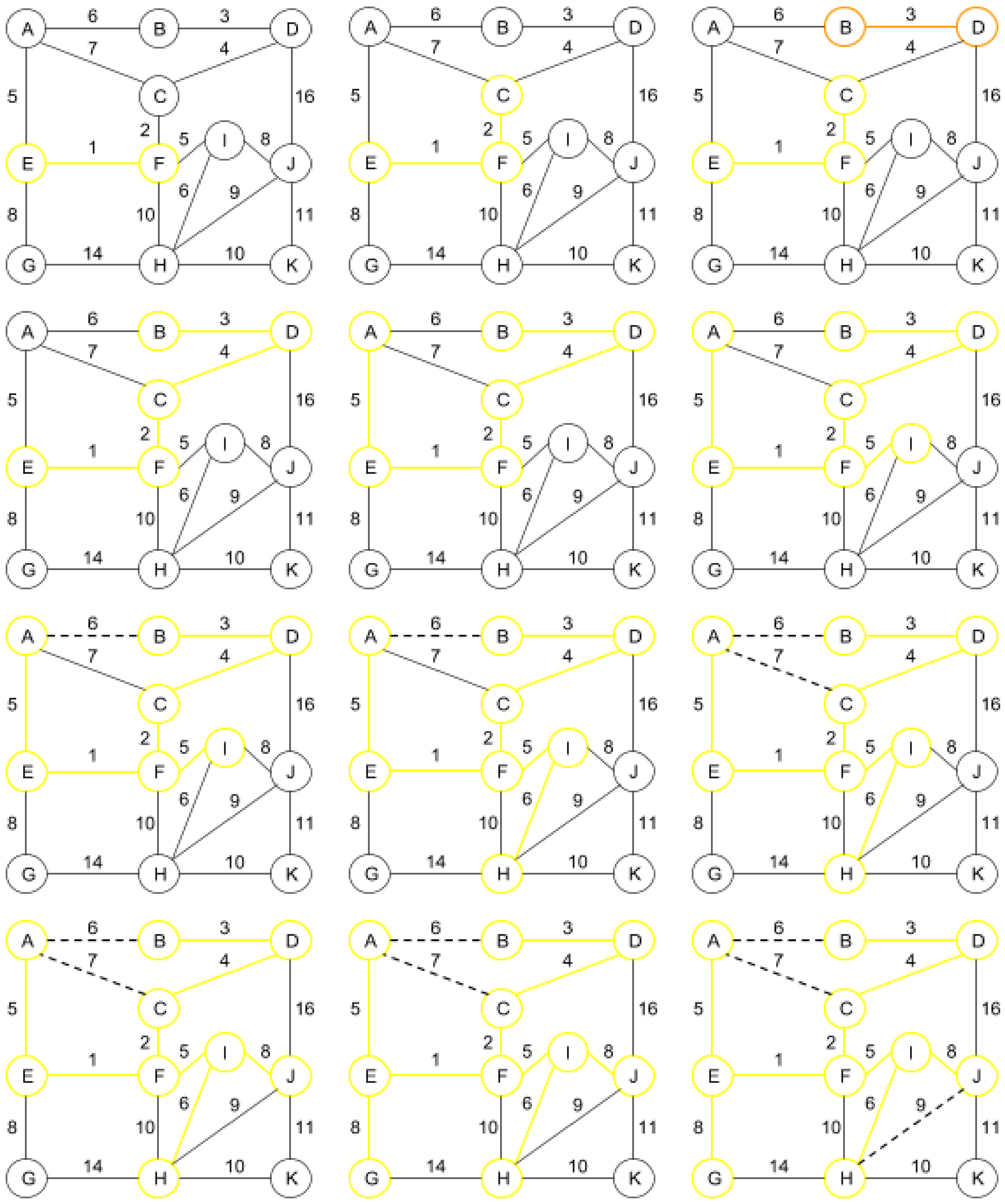
CS 1332 - Week 15

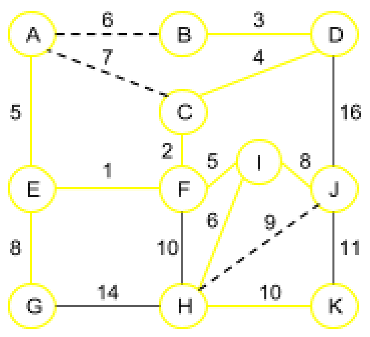
Monday

Topics

I. Kruskal's continued

   A. Example 2 (dashed edge was dequeued but not added to MST):





Done! The MST is valid because it has 10 edges.

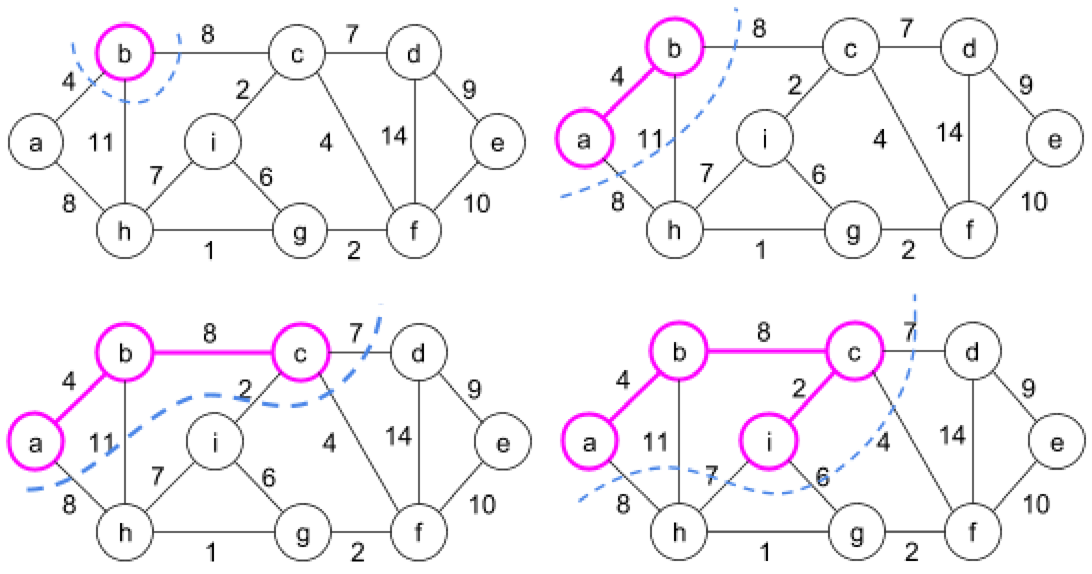
II. Prim's

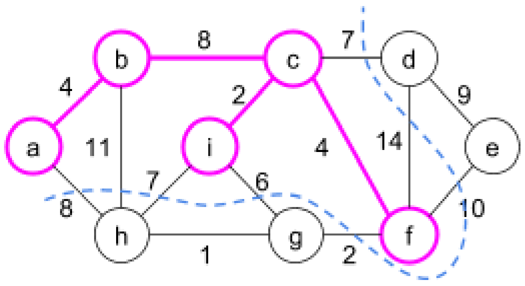
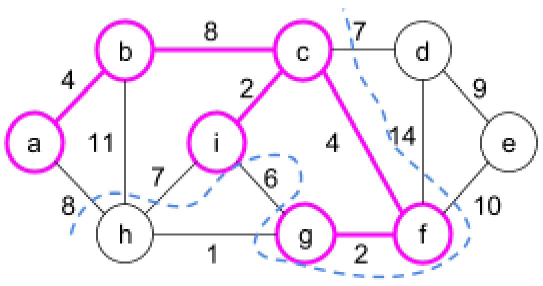
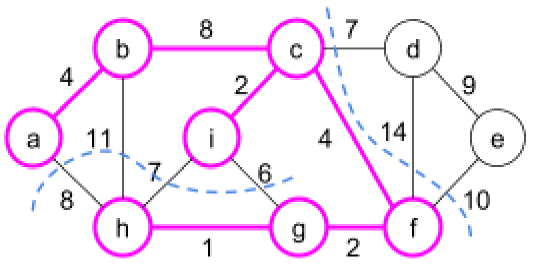
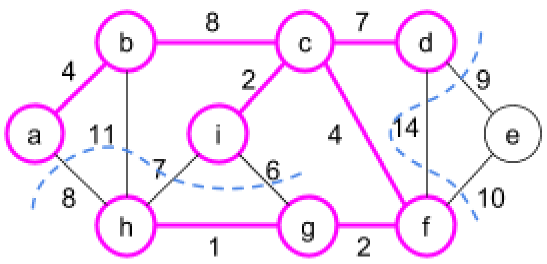
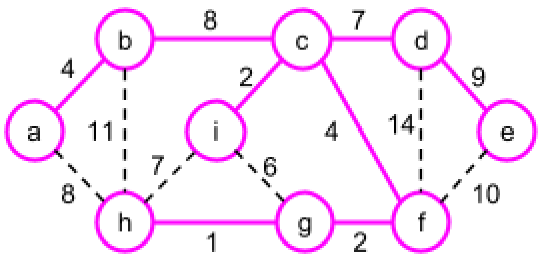
A. Premise - build a MST using a process similar to Dijkstra's (instead of adding aggregate paths to the priority queue like in Dijkstra's, you just add edges)

B. Algorithm:

1. Choose a starting vertex and add all of its incident edges to the PQ.
2. While the PQ is not empty and the visited set is less than the vertex set:
    a) Remove edge e from the priority queue
    b) If e.v is not visited → add e to MST and add e.v to visited
        (1) Add each incident edge to the PQ if its adjacent vertex has not yet been visited
3. Check MST validity

C. Example (starting at B): edges on the blue "cut" represent edges in the PQ



Done! MST is valid.

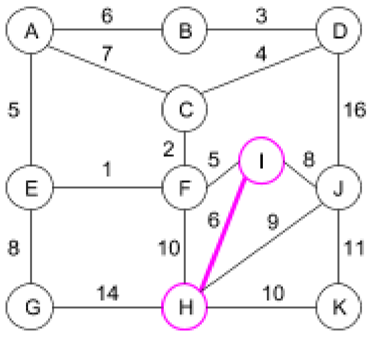
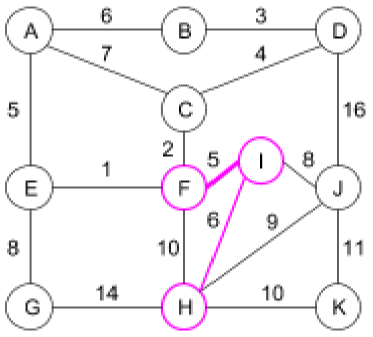
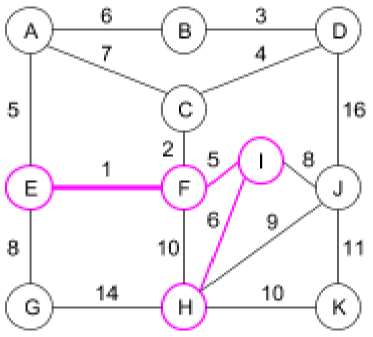
Wednesday

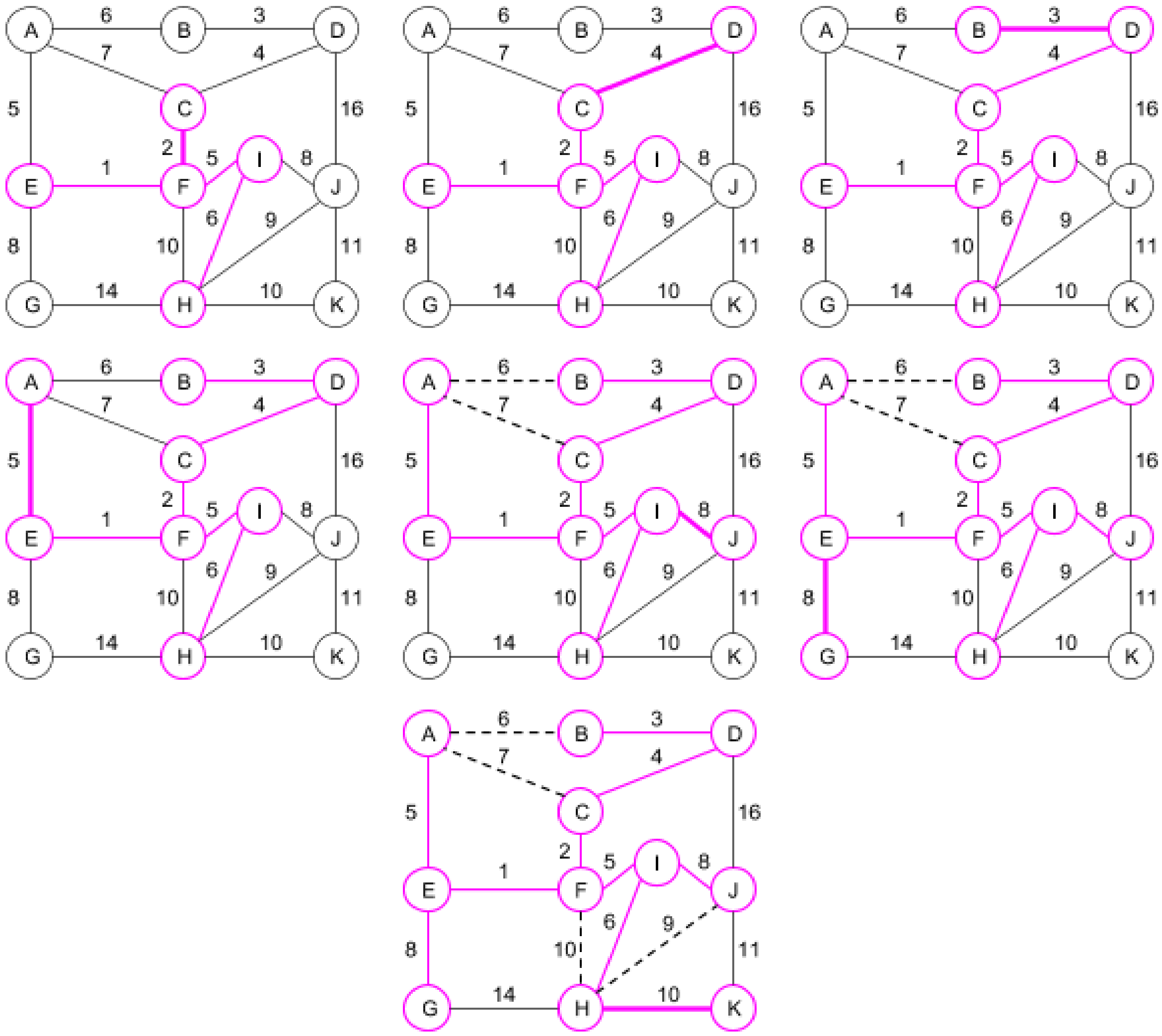
Topics

I. Prim's continued

  A. DIY example - pick your own start vertex and create an MST using Prim's

    1. For the start vertex H:



Done! MST is valid.

Friday

Topics

I. Dynamic Programming
   A. Premise - solve a large problem by solving smaller, repeated subproblems
   B. Components of dynamic programming:
      1. Identifies small repeated problems
      2. Establishes an order of solving
      3. Finds a solution to a large problem based on the subproblems

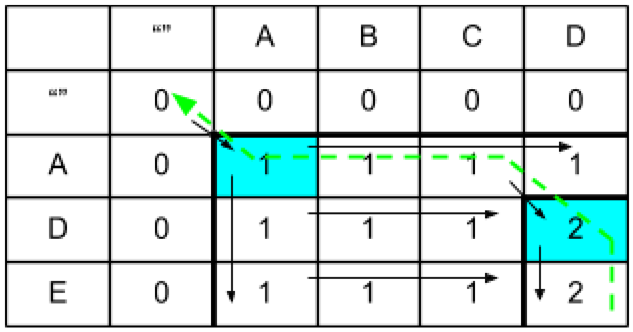
II. Dynamic Programming - Longest Common Subsequence (LCS)
   A. Subsequence - a sequence of characters that occur in a string in their same relative order, but aren't necessarily contiguous like in a sub*string*
      1. Eg. for the string APPLE:
         a) Valid substrings & subsequences: APPL, PPLE, APP
         b) Invalid substrings, valid subsequences: APE, ALE, APPE
         c) Invalid substrings & subsequences: LEAP, AEP, APLP

B. How do we find the LCS of strings s1 and s2 with dynamic programming?
   1. Using a 2D array to store the length of the LCS for substrings of s1 and s2, we start by comparing only the first letter of s1 to the first letter of s2 and see if we can increment the length of the LCS.
   2. Then, we'll gradually the extend the substrings of s1 and s2 that we are looking at and see if we can build off the current common subsequence.

C. Algorithm
   1. Create a 2D array (L) with s1.length + 1 rows and s2.length + 1 columns
   2. Fill all of row 0 and column 0 with 0s, the character in row and column 0 represents the empty string; the length of the LCS of anything with the empty string is 0.
   3. Starting from L[1][1] and for each cell L[i][j] moving across the rows, compare s1[i] and s2[j]:
      a) If the characters are equal → L[i][j] = L[i-1][j-1] + 1
      b) Else → L[i][j] = max(L[i-1][j],L[i][j-1])
         (1) If they are equal, consistently take from the left or above; this is necessary for tracing back.
         (2) Taking from above vs. left can yield different LCSs.
   4. When the array is filled, the value at L[s1.length][s2.length] is the length of the LCS; to find the actual LCS, start at that bottom right corner and trace back through where you got each number from (left, above, or diagonal). When you hit a value that came from the diagonal, record the character.

D. Example 1 - find the LCS of ADE and ABCD

	""	A	B	C	D
""	0	0	0	0	0
A	0	1	1	1	1
D	0	1	1	1	2
E	0	1	1	1	2



	""	A	B	C	D
""	0	0	0	0	0
A	0	1	1	1	1
D	0	1	1	1	2
E	0	1	1	1	2

The LCS is "AD"

E. Example - find the LCS of TOGAS and TACOCAT

		T	A	C	O	C	A	T
	0	0	0	0	0	0	0	0
T	0	1	1	1	1	1	1	1
O	0	1	1	1	2	2	2	2
G	0	1	1	1	2	2	2	2
A	0	1	2	2	2	2	3	3
S	0	1	2	2	2	2	3	3

The LCS is “TOA”