Algorithm Design and Analysis

วิชาบังคับก่อน: 204251 และ 206281
ผู้สอน: ตอน 1 ผศ. เบญจมาศ ปัญญางาม เรียน ห้อง 207
ตอน 2 ผศ. ดร. จักริน ชวชาติ เรียน ห้อง 209

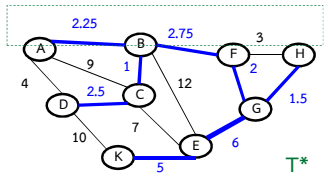
บทที่ 8
อัลกอริทึมกรีดดี
(Greedy algorithms)
Part II

MST : Cut & Cycle Properties

- Both Prim's and Kruskal's algorithms
  - Greedy algorithms using two general properties:
    - The cut property and the cycle property.

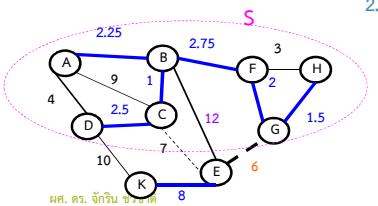
MST : Cut & Cycle Properties

- T* is MST, T* has two properties:



T*

1. Cut property
   - Let S be any subset of nodes, and let a be the min cost edge with exactly one endpoint in S.
   - Then the MST T* contains a.
2. Cycle property
   - Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST T* does not contain b.



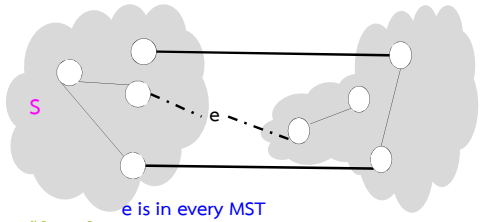
MST : Cut Property

- Simplifying assumption. All edge costs c_e are distinct.

Let S be any subset of nodes, and

let e be the min cost edge with exactly one endpoint in S. Should e be in every MST?

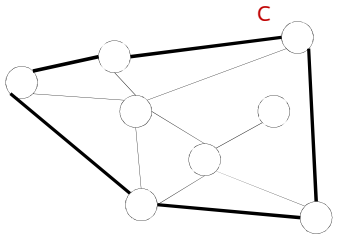
Yes ➔ Cut Propery



e is in every MST

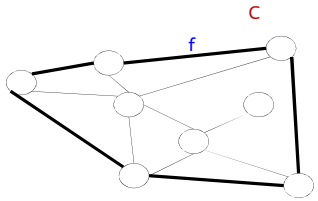
MST : Cycle Property

- Simplifying assumption. All edge costs c_e are distinct.
  - Let C be any cycle, does a MST exist that has all of C's edges?
  - No



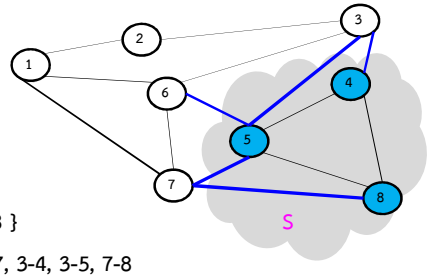
MST : Cycle Property

- Simplifying assumption. All edge costs c_e are distinct.
  - Let C be any cycle, does a MST exist that has all of C's edges?
  - No
  - Which one should be not in the MST?
  - The max cost ➔ Cycle Propery



MST : Cut and cutset

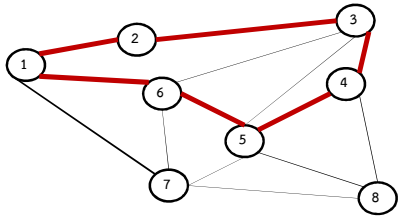
- A cut (S) is a subset of nodes.
- A cutset (D) of a cut S is the subset of (cut) edges with exactly one endpoint in S.



- A cut set ➔ S = { 4, 5, 8 }
- A cutset ➔ D = 5-6, 5-7, 3-4, 3-5, 7-8

MST : Cycles and Cuts

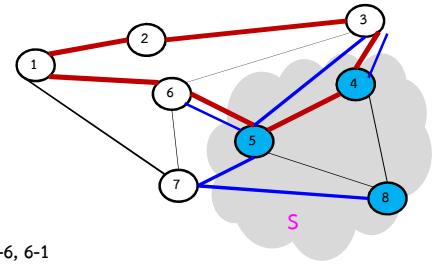
- Cycle.
  - Set of edges the form a-b, b-c, c-d, ..., y-z, z-a.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

MST : Cycle-Cut Intersection

- Consider the intersection of a cycle and a cutset.
- How many edges are there in such an intersection? (1, 2, odd, even ?)

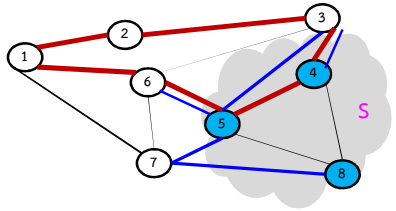


Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

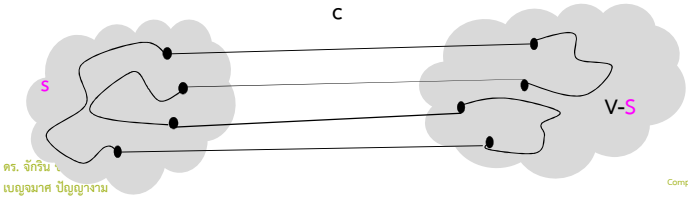
MST : Cycle-Cut Intersection

- Claim. A cycle and a cutset intersect in an even number of edges.

Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6



Proof. Walk along cycle from a node s∈S: for every edge leaving S, there should (first) be an edge to a node in S before returning to s.



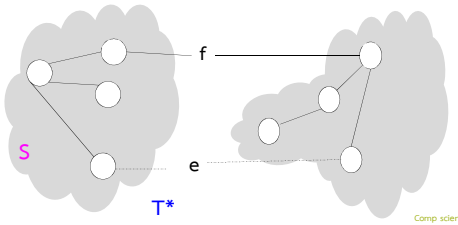
MST : Cut property

- Simplifying assumption. All edge costs c_e are distinct.
- Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e.

Proof. (by contradiction)

- Suppose e does not belong to T*, and let's see what happens.

This is a contradiction

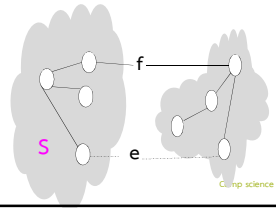


MST : Proof of Cut property

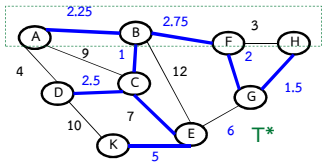
- Simplifying assumption. All edge costs c_e are distinct.
- Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e.

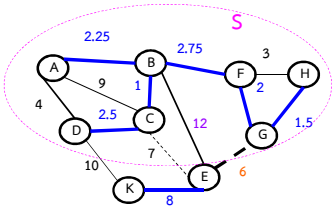
Proof. (by contradiction)

- Suppose e does not belong to T*, and let's see what happens.
- Adding e to T* creates a cycle C in T*.
- Edge e is both in the cycle C and in the cutset D corresponding to S , there exists another edge, say f, that is in both C and D.
- T' = T* U { e } - { f } is also a spanning tree.
- Since $c_e < c_f$, cost(T') < cost(T*).
- This is a contradiction.



MST : Proof of Cut property





- Simplifying assumption:
  - All edge costs c_x are distinct.
- Suppose T* is MST
- Let Edge x = (G,E) be not in T* and add (G,E) creates a cycle α={B,C,E,G,F} in T*
  - edge x is in both cycle α and tree S
- ➔ There exists another edge, y= (E,C), that is in both α and S

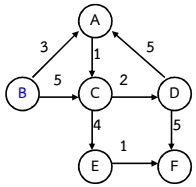
T'=T*$\cup${x} – { y } is also a spanning tree

And if $c_x < c_y$, then Cost(T') < Cost(T*)

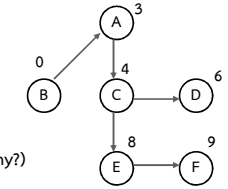
This is a contradiction

The Shortest Path Problem

- To find the shortest path from a given node to *all* other reachable nodes in a graph
- Dijkstra's algorithm builds up a *tree:* there is a path from each node back to the starting node
- For example, in the following graph, we want to find shortest paths from node B



- Edge values in the graph are weights
- Node values in the tree are *total* weights
- The arrows point in the *right direction* for what we need (why?)



The Shortest Path Problem : Dijkstra's algorithm

- Initially,
  - Mark the given node as *known* (path length is zero)
  - Set the distance in each neighboring node equal to the *cost* (length) of the out-edge
- Repeatedly (until all nodes are known),
  - Find an unknown node containing the smallest distance and mark the new node as known
  - For each node adjacent to the new node, examine its neighbors to see whether their estimated distance can be reduced (distance to known node plus cost of out-edge)

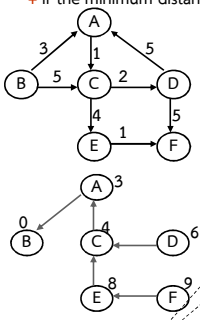
The Shortest Path Problem : Dijkstra's algorithm

```
1.  For all v∈V do d[v]=∞
2.  S := ∅;          d[s]=0;
3.  Q := V[G];
4.  while Q ≠ ∅ do
5.        u := Extract-Min(Q);
6.        S := S ∪ {u};
7.        for each v ∈ Adj[u] do
8.          if d[v]>d[u]+w(u, v)
9.            d[v]=d[u]+ w(u, v)
10.       endif
11.       endfor
12. endwhile
```

- Maintains a set S of vertices whose shortest path from s has been determined.
- Repeatedly selects u in V–S with minimum SP estimate (greedy choice).
- Store V–S in priority queue Q.

The Shortest Path Problem : Dijkstra's algorithm

+ if the minimum distance is known ➔d



S={ B A C D E F }

node	init'ly	1	2	3	4	5	6
A	inf	3B	+3B	+3B	+3B	+3B	+3B
B	0	+0	+0	+0	+0	+0	+0
C	inf	5B	4A	+4A	+4A	+4A	+4A
D	inf	inf	inf	6C	+6C	+6C	+6C
E	inf	inf	inf	8C	8C	+8C	+8C
F	inf	inf	inf	inf	11D	9E	+9E

priority queue
Q={B A C D E F }

B A C D E F ➔ A C D E F
A C D E F ➔ C D E F
C D E F ➔ D E F
D E F ➔ E F
E F ➔ F
F ➔ ∅

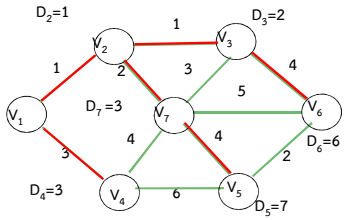
➔d(v)= d(u)+w(u,v)= 3+1

ตรวจสอบ d(v)>d(u)+w(u,v) สำหรับ v➔ A,C

➔d(v)= d(u)+w(u,v_1)= 0+3

➔d(v)= d(u)+w(u,v_2)= 0+5

The Shortest Path Problem : Dijkstra's algorithm

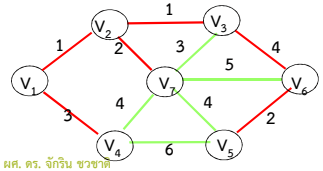


S= {1,2,3,4,7,6,5}

	h=1	h=2	h=3	h=4	h=5	h=6
V_2	1	1	1	1	1	1
V_3	∞	2	2	2	2	2
V_4	3	3	3	3	3	3
V_5	∞	∞	∞	9	7	7
V_6	∞	∞	6	6	6	6
V_7	∞	3	3	3	3	3

This is not a minimum weight spanning tree

A MST for this graph ➔



Q ={V_2 V_3 V_4 V_5 V_6 V_7 }
={V_3 V_4 V_5 V_6 V_7}
={V_4 V_5 V_6 V_7}
={V_7 V_5 V_6 }
={V_6 V_5}
={V_5}

Resource scheduling

- 9 jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes
- 3 processors on which you can run these jobs

Greedy choice : The longest-running jobs first, on whatever processor is available

P1	20	10	3
P2	18	11	6
P3	15	14	5

- Time to completion: 18 + 11 + 6 = 35 minutes
- This solution isn't bad, but we might be able to do better

Resource scheduling

- 9 jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes
- 3 processors on which you can run these jobs

Greedy choice : The shortest job first , on whatever processor is available

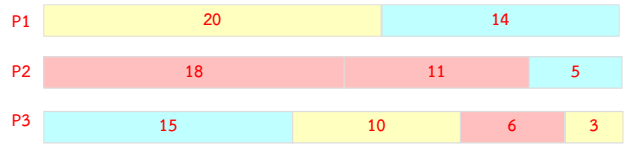
P1	3	10	15
P2	5	11	18
P3	6	14	20

- That wasn't such a good idea; time to completion is now 6 + 14 + 20 = 40 minutes
- Note, however, that the greedy algorithm itself is fast
  - All we had to do at each stage was pick the minimum or maximum

Resource scheduling

- Better solutions do exist:

An optimum solution



- This solution is clearly optimal
- Clearly, there are other optimal solutions
- How do we find such a solution?
  - One way: Try all possible assignments of jobs to processors
  - Unfortunately, this approach can take exponential time

Activity-Selection Problem

- Job scheduling time constrain and Single processor
- Input : a set S of n activities, a_1, ..., a_n

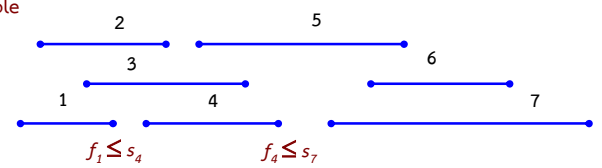
  s_i = start time of activity i

  f_i = finish time of activity i

  - Output: max-size subset A of compatible activities

    •Two activities are compatible, if their intervals don't overlap

example



Activity Selection: Optimal Substructure

- Assume activities are sorted by finishing times.
  - $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Suppose an optimal solution includes activity a_k.
  - This generates two subproblems.
  - Selecting from a_1, ..., a_{k-1}, activities compatible with one another, and that finish before a_k starts (compatible with a_k).
  - Selecting from a_{k+1}, ..., a_n, activities compatible with one another, and that start after a_k finishes.
  - The solutions to the two subproblems must be optimal.

Activity Selection: A Greedy Algorithm

- Make the greedy choice and solve top-down.
- May have to preprocess input to put it into greedy order.
  - Example: Sorting activities by finish time,
- So actual algorithm is simple
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities

Coin Counting Problem : Greedy strategy

Goal: To find the minimum number of coins to make any amount k

- Two elements
  1. Optimization problem :
     - need minimum number of coins
  2. greedy choice property

- Make a greedy choice
  - Select the highest valued coin available
    - which is not greater than the amount to be paid

Coin Counting : Greedy strategy

- In some (fictional) monetary system,

  $5 $1, $.25 $.10 $.1
- Using a greedy algorithm to find the minimum number of coins for changing of $6.34
  - you would get
- One coin of $5, One coin of $1
- One coin of $.25, One coin of $.10, four coins of $.01
- The greedy algorithm always gives the optimum solution

Homework3: Implement a program to solve the Coin Counting Problem by Greedy method
ส่งที่ bpanyangam@gmail.com

Coin Counting Problem : Greedy strategy

- A failure of the greedy algorithm
  - In some (fictional) monetary system, coin value :1, 7,10
  - Using a greedy algorithm to find the minimum number of coins for changing of 15
    - You get: One coins of 10 and Five coins of 1
      - Requires six coins
  - A better solution would be to use only 3 coins two of 7 and one of 1
- The greedy algorithm results in a solution, but not in an optimal solution

Coin Counting Problem :Optimal Substructure

- Let count(S[], m, n)
  - function to count the number of coins,
  - To count total number solutions, we can divide all set solutions in two sets.
    1. Solutions that do not contain m^{th} coin (or S_m)
       - count(S[], m-1, n)
    2. Solutions that contain at least one S_m
       - count(S[], m, n-Sm)

count(S[], m, n) = count(S[], m-1, n) + count(S[], m, n-Sm)

Coin Counting Problem :Overlapping sub problems

- recursive implementation

```
count( n, m )
if n = 0 then  return 1
if n < 0  then return 0 //n < 0 ➔ no solution
if m < 1 and n > 1 then return 0  //no coins and n > 0➔no solution
return count( n, m - 1 ) + count( n - S[m], m )
```

➔many subproblems being called more than once

Applications of the Greedy Strategy

- Optimal solutions:
  - Some instances of change making
  - Minimum Spanning Tree (MST)
  - Single-source shortest paths
  - Huffman codes
- Approximations:
  - Traveling Salesman Problem (TSP)
  - Knapsack problem
  - Other optimization problems

For some problems, yields an optimal solution for every instance.

For most, does not but can be useful for fast approximations