# *Greedy Methods*

## *Problems whose solutions can be "ranked"*

| | Travel | Investment | Course selection |
|---|---|---|---|
| Feasible solutions | stay on highway, finish in x days | don't spend more than one has | finish in 4 years |
| Optimal solutions | shortest distance, minimum time | maximum returns, minimum risks | best combination of depth and breadth |
| Decisions | which highways to take | invest or not in a portfolio | take a course or not |

- Decisions can be made
  - one at a time, **without** backtracking
  - Greedy method
  - *Which decisions to make next*?
  - How to guarantee optimality?
- Try many (all) possible combinations and choose one which is the best
  - Dynamic programming
  - How to test multiple solutions efficiently?

# *The Greedy Method*

- Input $n$ elements stored in an array $A(1:n)$
- Procedure Greedy
  - Solution = *NULL*
  - for $i=1$ to $n$ do
    - x = SELECT($A$)
    - if FEASIBLE(Solution, x)
    - then Solution = UNION(Solution, x)
    - endif
  - enddo
  - return (Solution)

- A sequence of *n* decisions w.r.t *n* inputs
- *SELECT:* select one of the remaining decisions to make according to some *optimization* measure
  - once a decision is made, it will ***not*** become invalid at a later time
  - *optimization* should be based on the partial solutions built so far
- *FEASIBLE:* whether the partial solution satisfies some preset constraints

- *Strategy*: construct feasible solutions one step at a time which optimize (minimize or maximize) a certain objective function
- *Make the obvious decisions first*!
- *Then try to show it is indeed optimal!*

# *Knapsack problem*

- Input:
  - a set of n objects $(P_i, W_i) \;\; i = 1, \ldots, n$
  - a knapsack of capacity $M$
- Output: fill the knapsack to maximize the total profit earned
- Feasibility constraint: $\sum_{i=1}^{n} W_i X_i \le M$
- Objective function: $\max \sum_{i=1}^{n} P_i X_i \quad 0 \le X_i \le 1$

## ❖ Example

$n = 3, M = 20$

$(P_1, P_2, P_3) = (25,24,15)$

$(W_1, W_2, W_3) = (18,15,10)$

| $(X_1, X_2, X_3)$ | $\sum_{i=1}^{n} W_i X_i$ | $\sum_{i=1}^{n} P_i X_i$ | |
|---|---|---|---|
| $(1, \frac{2}{15}, 0)$ | 20 | 28.2 | largest increase in profit |
| $(0, \frac{2}{3}, 1)$ | 20 | 31 | smallest increase in weight |
| $(0, 1, \frac{1}{2})$ | 20 | 31.5 | largest increase in profit to weight ratio |

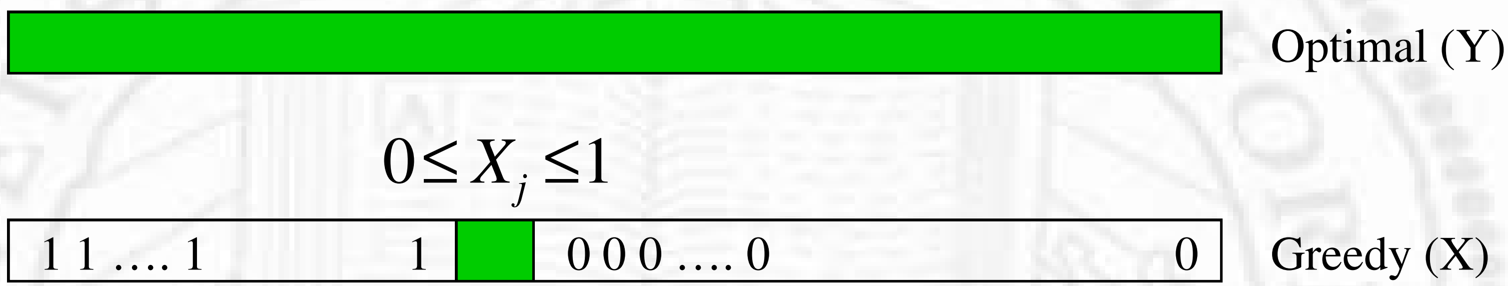
$$(\frac{P_1}{W_1}, \frac{P_2}{W_2}, \frac{P_3}{W_3}) = (1.39, 1.6, 1.5)$$

- For all three algorithms
  - decisions are made one object at a time
  - the *ordering* is determined by some optimization measure
    - Largest increase in profit
      - Include the remaining object of the largest profit
    - Smallest increase in weight
      - Include the remaining object of the smallest weight
    - Largest increase in profit/weight
      - Include the remaining object of the largest profit/weight
  - never backtrack
  - all greedy algorithms
  - not all guarantee optimal

- **Proposition**: Greedy selection based on maximizing profit to weight ratio gives the optimal result
- **General proof strategy**:
  - Assume that the greedy solution is $X = (X_1, X_2, ..., X_n)$
  - Assume that the optimal solution is $Y = (Y_1, Y_2, ..., Y_n)$
  - Then they better be different
  - Transform $Y$ into $X$ without decreasing the profit of $Y$

# ❖ Proof:

- Assume $\frac{P_1}{W_1} \geq \frac{P_2}{W_2} \geq \ldots \geq \frac{P_n}{W_n}$



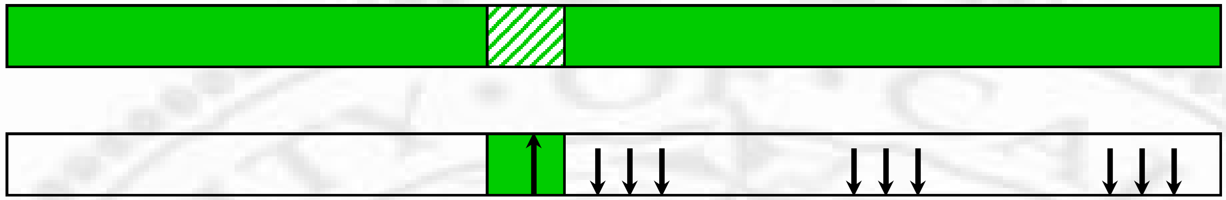

  – Let $k$ be the first index where $X$ and $Y$ differ

$$(i) \quad k < j \quad X_k = 1 \,\&\, Y_k \neq X_k \Rightarrow Y_k < X_k$$

$$(ii) \quad k = j \quad if\ Y_k > X_k\ then \sum W_i Y_i > M \Rightarrow Y_k < X_k$$

$$(iii) \quad k > j \quad X_k = 0 \,\&\, Y_k \neq X_k \Rightarrow Y_k > 0 \,\&\, \sum W_i Y_i > M,\ not\ possible$$

$Y_k < X_k$



Optimal (Y)

New optimal (Z)

$$\sum_{i=1}^{n} Z_i P_i = \underbrace{\sum_{i=1}^{n} Y_i P_i}_{\text{profit of Y}} \underbrace{+ (Z_k - Y_k) P_k}_{\text{increase of profit}} \underbrace{- \sum_{i=k+1}^{n} (Y_i - Z_i) P_i}_{\text{decrease of profit}}$$

$$= \sum_{i=1}^{n} Y_i P_i + (Z_k - Y_k) \frac{P_k}{W_k} W_k - \sum_{i=k+1}^{n} (Y_i - Z_i) \frac{P_i}{W_i} W_i$$

$$\geq \sum_{i=1}^{n} Y_i P_i + \{ \underbrace{(Z_k - Y_k) W_k}_{\text{increase in weight}} - \underbrace{\sum_{i=k+1}^{n} (Y_i - Z_i) W_i}_{\text{decrease in weight}} \} \frac{P_k}{W_k}$$

$$\geq \sum_{i=1}^{n} Y_i P_i$$

- $Z$ is also an optimal solution
- Either $Z=X$ (Done)
- Or not (Repeat the above procedure until $Z=X$)

- Time complexity
  - Sort the $n$ objects according to profit to weight ratio $O(nlogn)$
  - Scan down the sorted list

$$
\begin{array}{l}
\text{if } W_i \le \text{ remaing capaity then} \\
\quad X_i \leftarrow 1 \\
\quad \text{remaining capacity } -= W_i \\
\text{else} \\
\quad X_i \leftarrow \dfrac{\text{remaining capacity}}{W_i} \\
\quad \text{remaing capacity} \leftarrow 0 \\
\text{endif}
\end{array}
$$

– Complexity $O(nlogn)$

# *Optimal Storage on Tape*

- Input:
  - A set of *n* programs of different length
  - A computer tape of length *L*
- Output:
  - A storage pattern which minimizes the total retrieval time (*TRT*)
    - before each retrieval, head is repositioned at the front

$$TRT = \sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{i_k} \qquad I = i_1, i_2, \ldots i_n$$

- Objective function: minimize *TRT*
- Feasibility constraint: $\sum_{1 \le k \le n} l_{i_k} \le L$
- Example

$$n = 3, (l_1, l_2, l_3) = (5,10,3), L = 20$$

| *ordering* $(i_1, i_2, i_3)$ | | | *TRT* | |
|---|---|---|---|---|
| 1,2,3 | $5+$ | $5+10+$ | $5+10+3$ | $=38$ |
| 1,3,2 | $5+$ | $5+3+$ | $5+3+10$ | $=31$ |
| 2,1,3 | $10+$ | $10+5+$ | $10+5+3$ | $=43$ |
| 2,3,1 | $10+$ | $10+3+$ | $10+3+5$ | $=41$ |
| 3,1,2 | $3+$ | $3+5+$ | $3+5+10$ | $=29$ |
| 3,2,1 | $3+$ | $3+10+$ | $3+10+5$ | $=34$ |

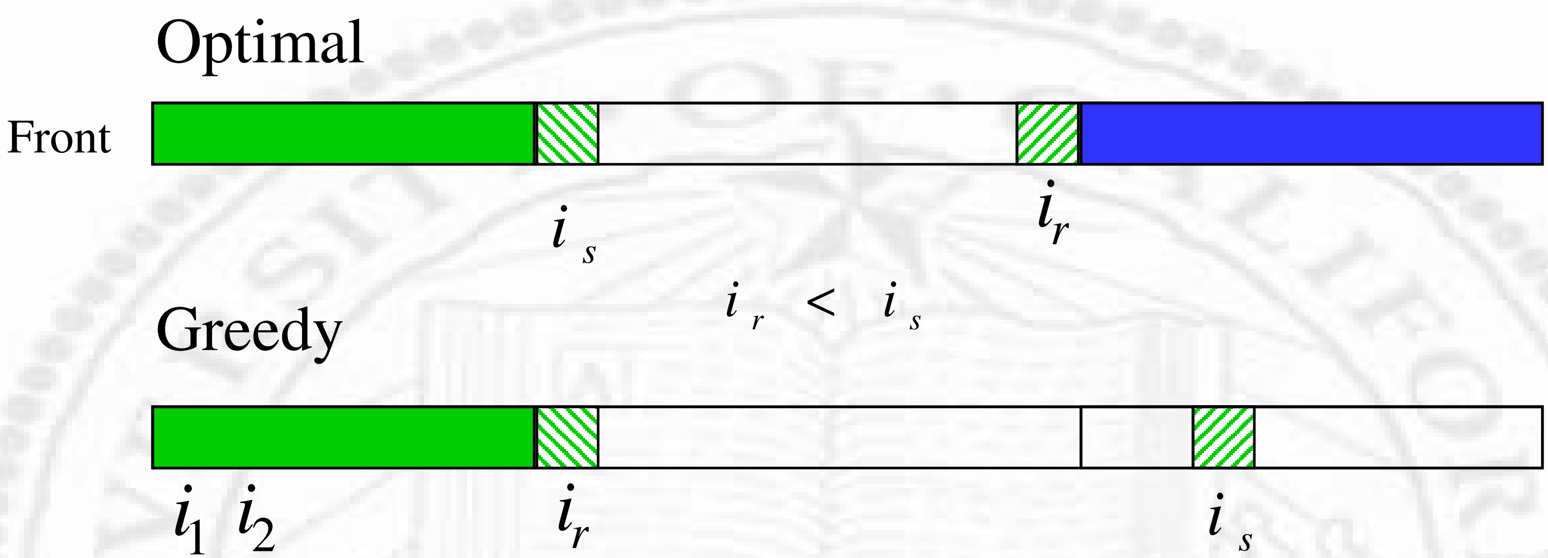
- *SELECT*: Select the program to store next which minimizes the increase in *TRT*


$$TRT_{old} = \sum_{1 \le j \le r} \sum_{1 \le k \le j} l_{i_k}$$

$$TRT_{new} = \sum_{1 \le j \le r+1} \sum_{1 \le k \le j} l_{i_k} = TRT_{old} + \sum_{1 \le k \le r+1} l_{i_k}$$

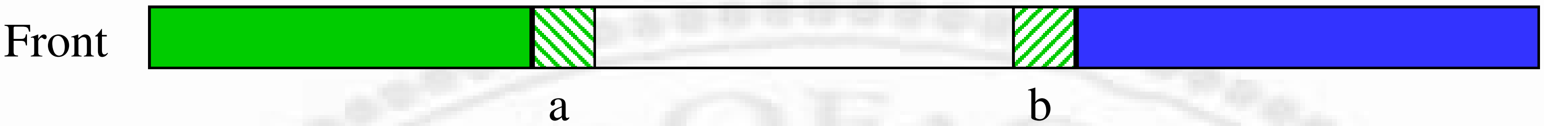
$$TRT_{new} - TRT_{old} = \sum_{1 \le k \le r+1} l_{i_k} = \underbrace{\sum_{1 \le k \le r} l_{i_k}}_{\text{fixed}} + \underbrace{l_{r+1}}_{\text{Currently shortest program}}$$

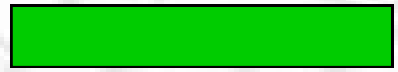
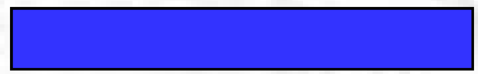
- Proof strategy
  - follow the same principle as in knapsack problem
    - there is a greedy solution
    - there is an optimal solution
    - they are different
    - line them up and they better differ in some storage locations
    - then make them the same (by swapping)
    - prove that the swapping does not reduce the optimality




- Swap $i_r$ and $i_s$ in the optimal solution

## ❖ Intuitively




- For programs stored in 
  - retrieval does not scan through either *a* or *b*
  - ordering of *a* and *b* not important
- For programs stored in 
  - retrieval scans through both *a* and *b*
  - ordering of *a* and *b* not important
- For programs stored in 
  - retrieval scans through *a* but not *b*
  - ordering of *a* and *b* is important

- **Proposition**: The storage pattern with nondecreasing length order produces the smallest *TRT*

$$
\begin{aligned}
\sum_{1\le j\le n}\sum_{1\le k\le j} l_{i_k} &= l_{i_1} \\
&\quad + l_{i_1} \quad + l_{i_2} \\
&\quad + l_{i_1} \quad + l_{i_2} \quad + l_{i_3} \\
&\quad \ldots \quad \ldots \quad \ldots \\
&\quad + l_{i_1} \quad + l_{i_2} \quad + l_{i_3} \quad + \ldots \quad + l_{i_n} \\
&= \mathrm{n} l_{i_1} + (n-1) l_{i_2} + (n-2) l_{i_3} + \ldots + l_{i_n}
\end{aligned}
$$

$$
= \sum_{1\le k\le n} (n-k+1) l_{i_k}
$$

- If prog. *a* and prog. *b* are out of order, then swap them should reduce the *TRT*

$$TRT_{old} = \sum_{\substack{k \\ k \neq a \\ k \neq b}} (n-k+1) l_{i_k} + (n-a+1) l_{i_a} + (n-b+1) l_{i_b}$$

$$TRT_{new} = \sum_{\substack{k \\ k \neq a \\ k \neq b}} (n-k+1) l_{i_k} + (n-a+1) l_{i_b} + (n-b+1) l_{i_a}$$

$$TRT_{old} - TRT_{new} = (n-a+1)(l_{i_a} - l_{i_b}) + (n-b+1)(l_{i_b} - l_{i_a})$$

$$= (b-a)(l_{i_a} - l_{i_b}) > 0$$

- Time complexity: *O(nlogn)* for sorting

# *Optimal Merge Pattern*

- Input: a set of files of different lengths
- Output: an optimal sequence of *two-way* merges to obtain a sorted files

$F_i, 1 \le i \le n,$ of length $q_i$

merge files $F_i$ and $F_j$ requires $O(q_i + q_j)$ time

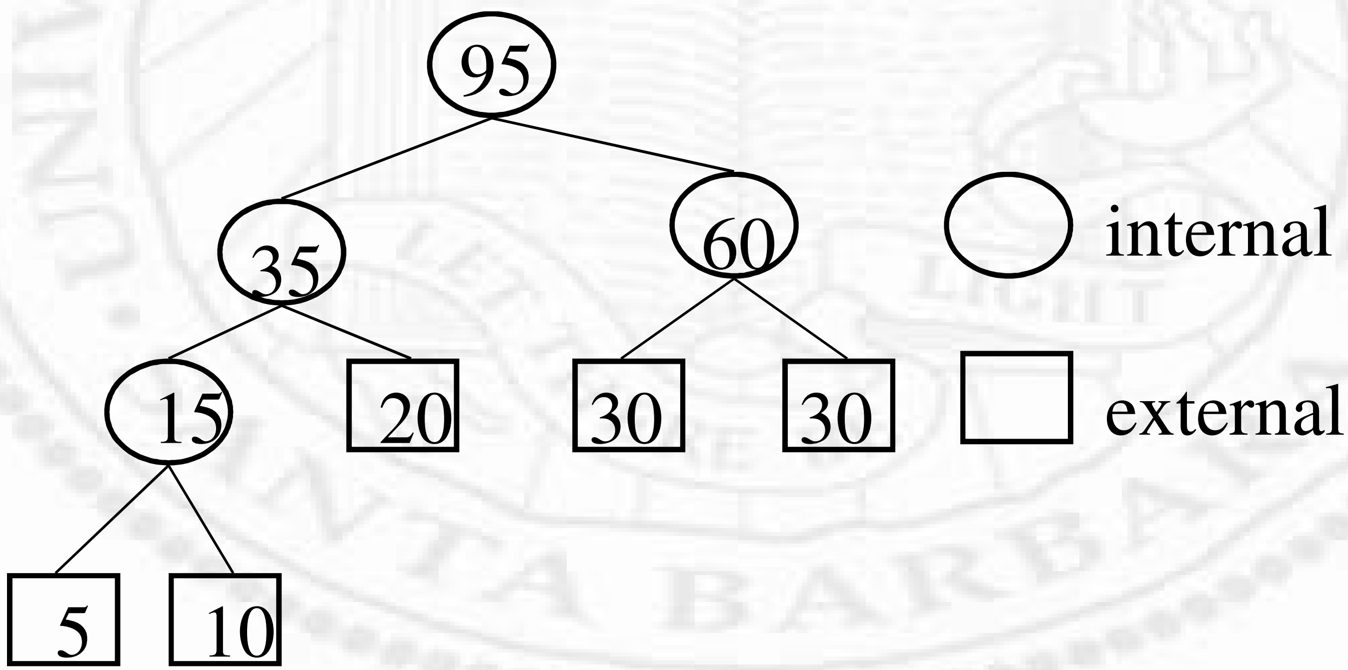
- Example

$$n = 3, (q_1, q_2, q_3) = (30, 20, 10)$$

| *ordering* | cost |
|---|---|
| 1,2,3 | $50 + 60 = 110$ |
| 1,3,2 | $40 + 60 = 100$ |
| 2,1,3 | $50 + 60 = 110$ |
| 2,3,1 | $30 + 60 = 90$ |
| 3,1,2 | $40 + 60 = 100$ |
| 3,2,1 | $30 + 60 = 90$ |

- Programs (files) stored on a tape (already merged together) may affect the access times (the merge times) of new programs (files) to be stored (merged)
- *SELECT*: At each step, merge two smallest files

- Binary merge tree
  - Distance from an external node to root = # of times a file is involved in merging
  - Total # of record moves for file *i* $d_i q_i$
    - external path length to reach node *i*
  - Total # of record moves for all files $\sum_{i=1}^{n} d_i q_i$
    - total external path length



# *Huffman Code*

- For data compression to save storage space and transmission bandwidth
- ASCII code uses *fixed*-length 8 bits/character code words, $O(8n)$ for storage and transmission
- Huffman codes uses *variable*-length code words depending on the *frequency of occurrence*

- Example

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| Frequency (thousands) | 45 | 13 | 12 | 16 | 9 | 5 |
| Fixed-length | 000 | 001 | 010 | 011 | 100 | 101 |
| Variable-length | 0 | 101 | 100 | 111 | 1101 | 1100 |

$$\text{fixed}-\text{length} \quad 100{,}100 \times 3 = 300{,}000 \text{ bits}$$

$$\text{variable-length } 45{,}000 \times 1 + 13{,}000 \times 3 + 12{,}000 \times 3$$

$$+ 16{,}000 \times 3 + 9{,}000 \times 4 + 5{,}000 \times 4 = 224{,}000 \text{ bits}$$

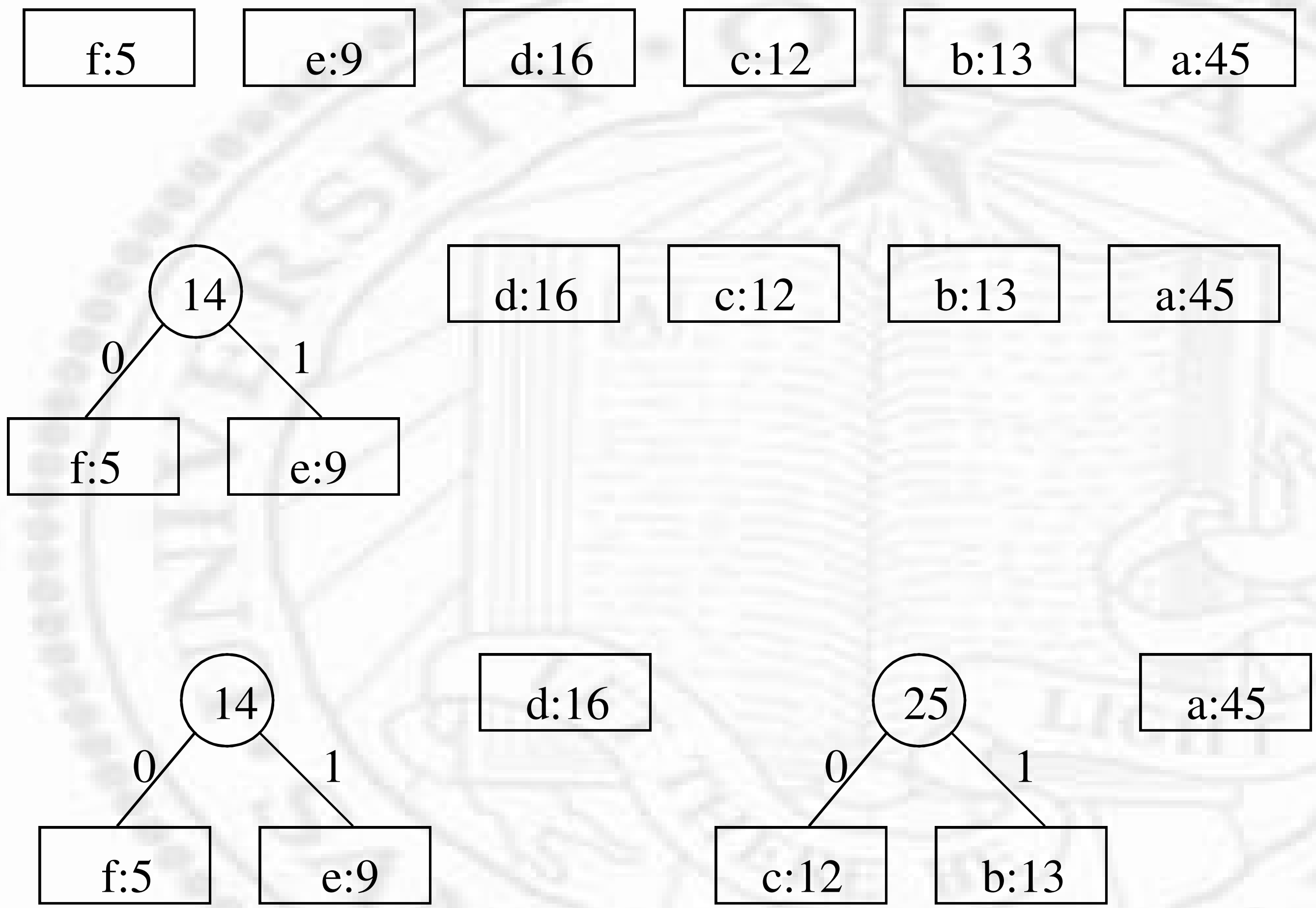
## Prefix codes

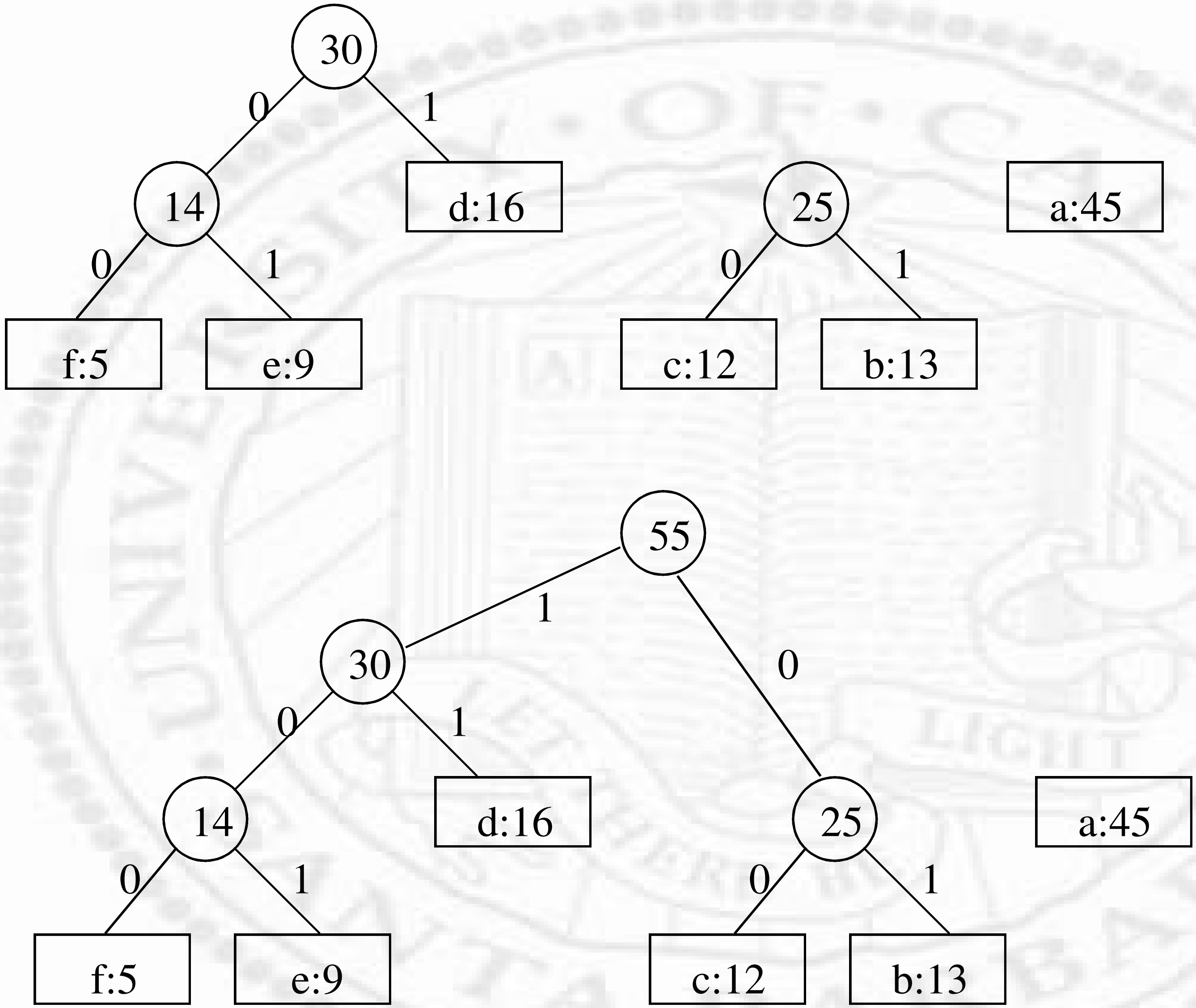
- no codeword is a prefix of some other codeword

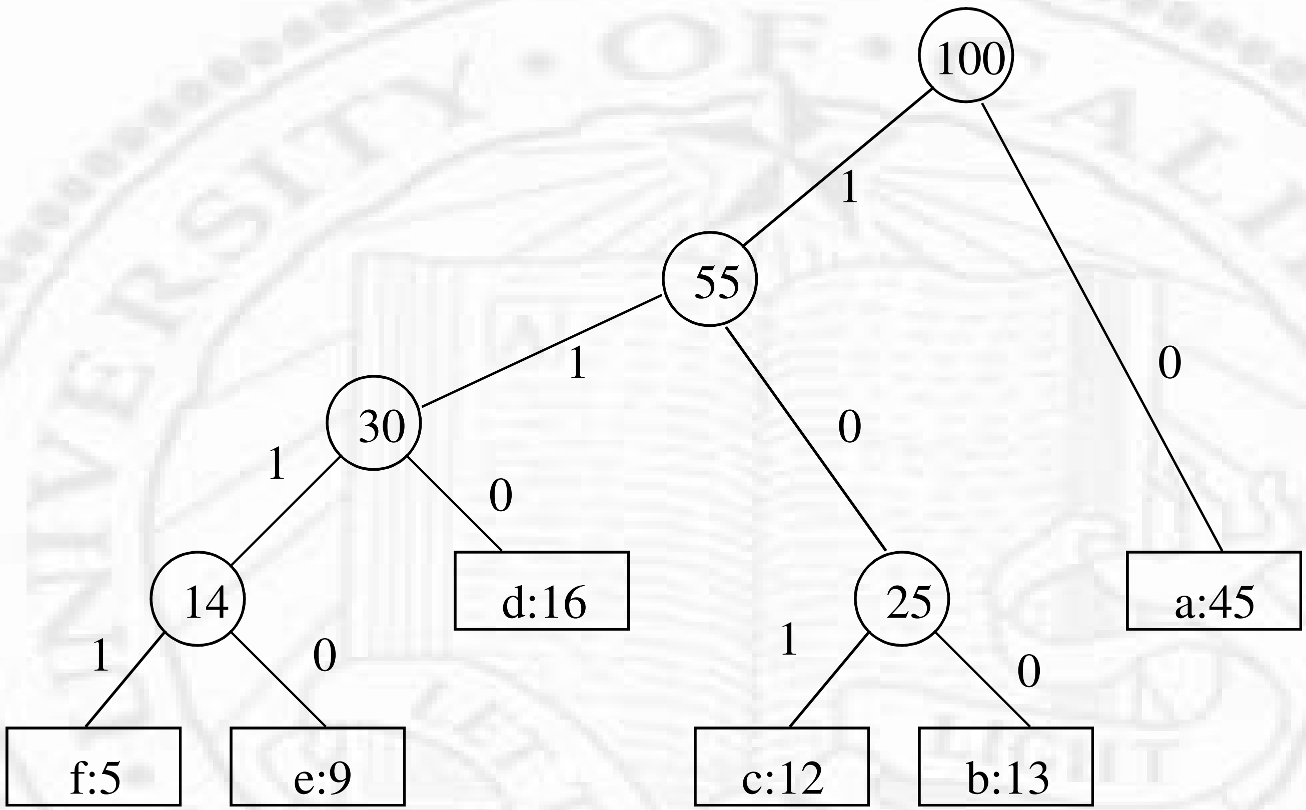
| | |
|---|---|
| 1010 | codeword a |
| 1010001 | codeword b |

… 110 …. 1010 …

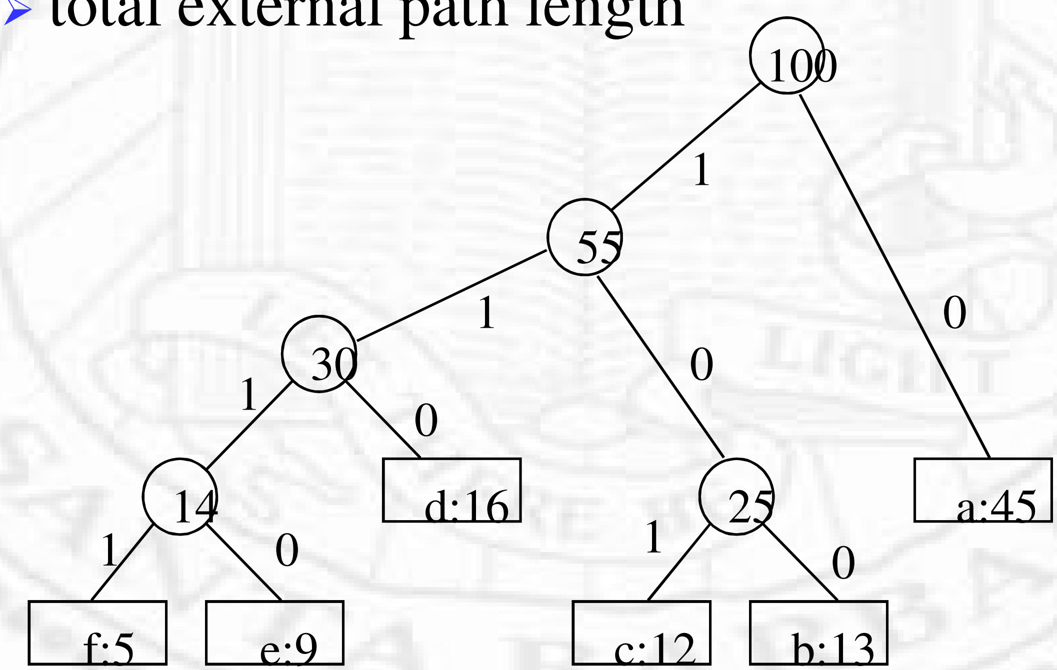
codeword a?
or beginning of codeword b?


f:5
e:9
d:16
c:12
b:13
a:45
14
0
1
f:5
e:9
d:16
c:12
b:13
a:45
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45


30
0
1
14
d:16
0
1
f:5
e:9
25
0
1
c:12
b:13
a:45
55
1
0
30
0
1
14
d:16
0
1
f:5
e:9
25
0
1
c:12
b:13
a:45

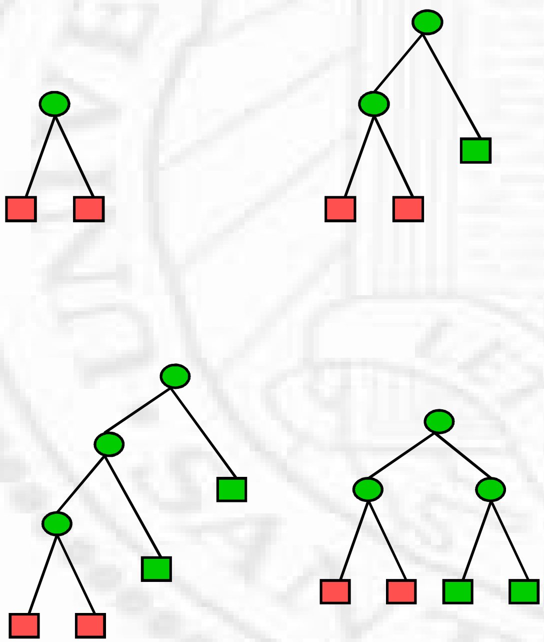

100
1
0
55
1
0
30
1
0
14
d:16
25
a:45
1
0
1
0
f:5
e:9
c:12
b:13

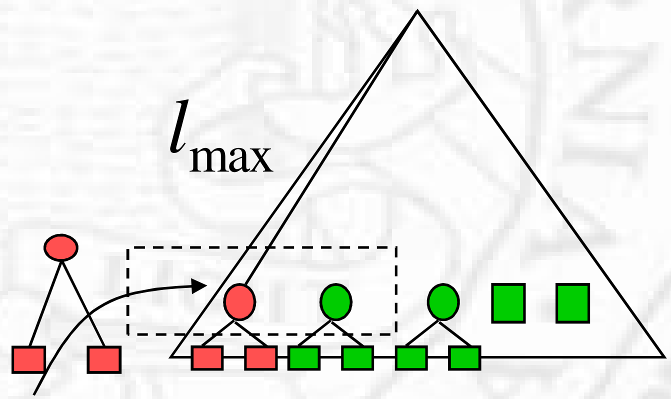
- Huffman code
  - Distance from an external node to root = # of bits in the code word
  - Total effort of sending i $d_i q_i$
    - external path length to reach node *i*
  - Total effort of sending all alphabets $\sum_{i=1}^{n} d_i q_i$
    - total external path length



# *An Important Fact*

- Using Huffman-tree rules, nodes that are merged first must have a longer path to the root than nodes that are merged later






$$p_1 + p_2 \leq p_i + p_j, \quad i, j > 2$$

No node can be merged more than once before p1+p2 is involved again

# *An Important Fact (cont.)*

- An iteration:
  - Between two successive merges involve p1
- In an iteration
  - No node can be involved in more than one merge
  - No node can increase in path length more than p1
- Hence, p1 must have the longest path length

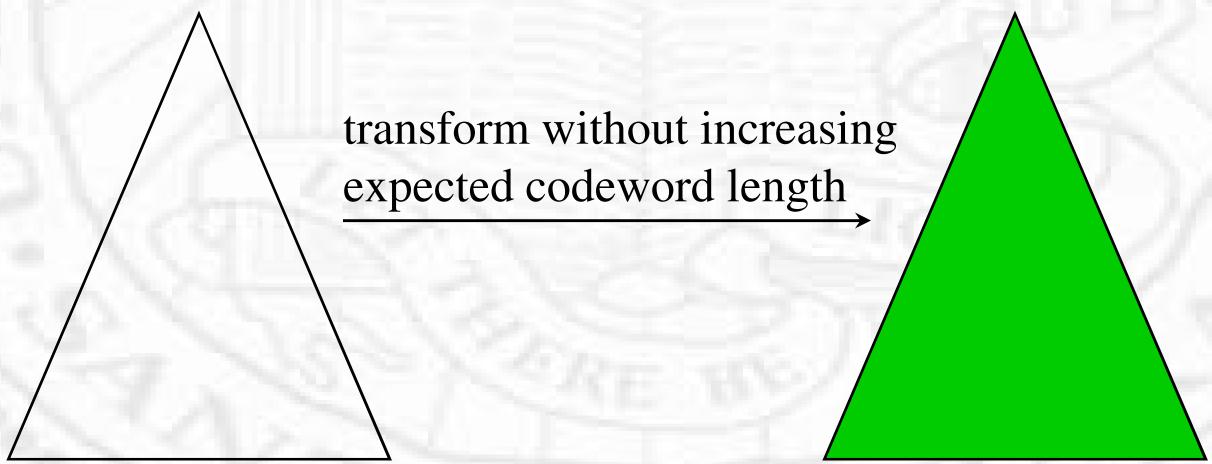
- **Proposition**: Huffman construction minimizes the expected codeword length

$$\sum_{i=1}^{n} p_i l_i$$

$p_i$ probability of occurance

$l_i$ codeword length

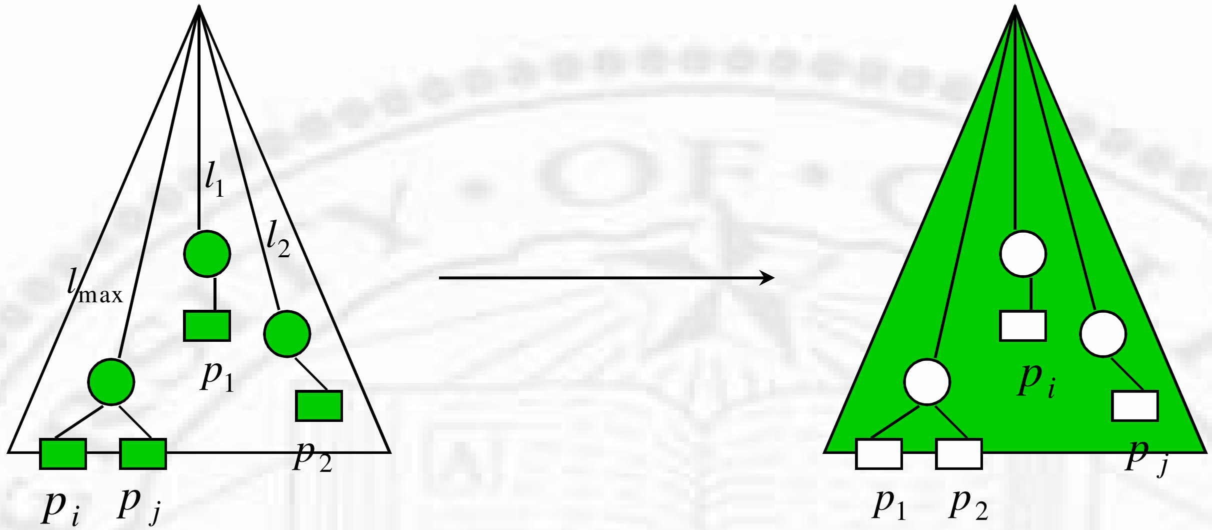
- **Proof:** $Assume\ that\ p_1 \leq p_2 \leq \ldots \leq p_n$




Optimal prefix-code tree

Huffman prefix-code tree

Optimal prefix-code tree

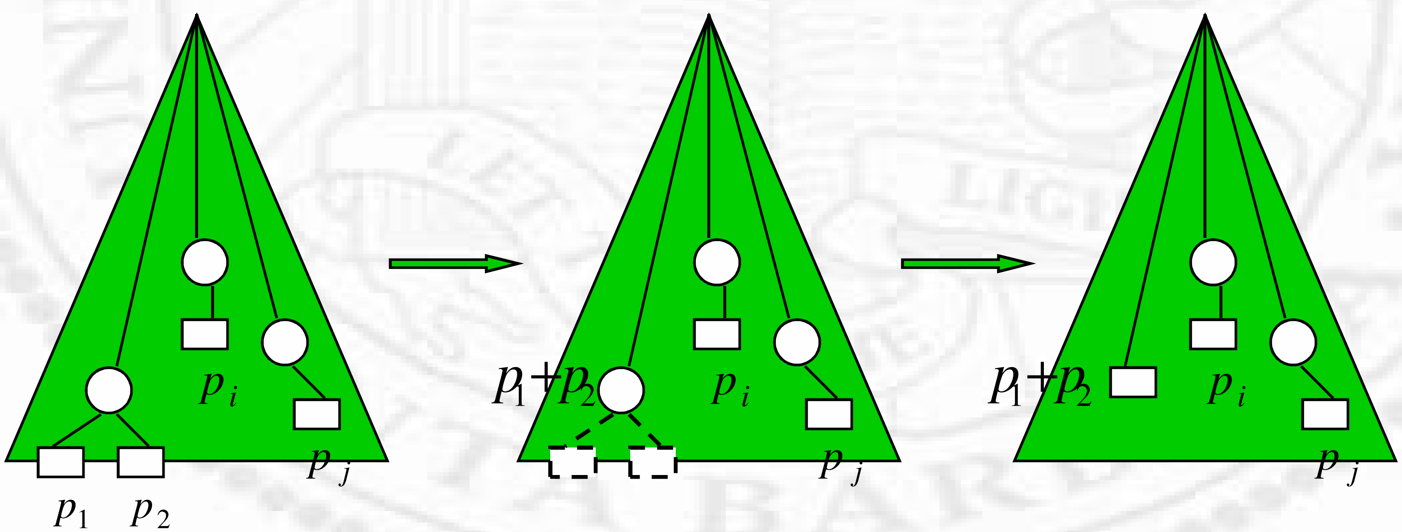



$$\sum_{k=1}^{n} p_k l_k = \sum_{k \neq 1,2,i,j} p_k l_k + l_{\max} p_i + l_{\max} p_j + l_1 p_1 + l_2 p_2$$

$$\sum_{k=1}^{n} p_k l_k' = \sum_{k \neq 1,2,i,j} p_k l_k + l_{\max} p_1 + l_{\max} p_2 + l_1 p_i + l_2 p_j$$

$$\sum_{k=1}^{n} p_k l_k - \sum_{k=1}^{n} p_k l_k'$$
$$= \underline{l_{\max} p_i} + \underline{l_{\max} p_j} + \underline{l_1 p_1} + \underline{l_2 p_2} - \underline{l_{\max} p_1} - \underline{l_{\max} p_2} - \underline{l_1 p_i} - \underline{l_2 p_j}$$
$$= (l_{\max} - l_1)(p_i - p_1) + (l_{\max} - l_2)(p_j - p_2) \geq 0$$

- Recursion
  - once p1 and p2 are moved to their right locations
  - merge them into a single node of p1+p2
  - now, greedy method will select from p1+p2, p3, …, pn the smallest two to merge
  - if that is not the case for optimal, then ...



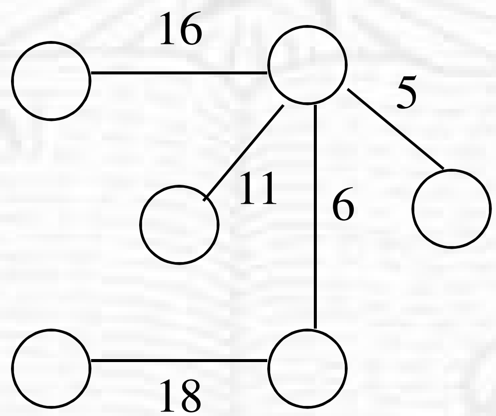
- Time complexity
  - with $n$ alphabets to code, exactly $n$-$1$ merges are needed
  - for each merge
    - find an least-frequently-used alphabet
    - find the next least-frequently-used alphabet
    - merge
    - put merged subtrees back into the list of subtrees
  - priority queue (heap) is ideal for this operation
  - $O(n)$ steps of *detelemin* and *insert* ($O(logn)$)
  - $O(nlogn)$

# *Minimum-Cost Spanning Tree*

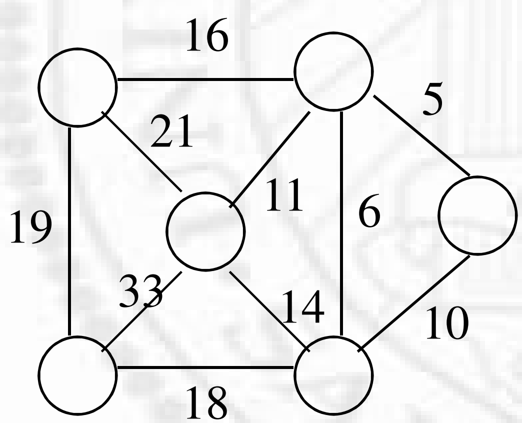
- Input: $G=(V,E)$, an undirected, labeled graph
- Output: $T=(V,E')$, a subgraph of G
  - includes all the vertices
  - is a tree

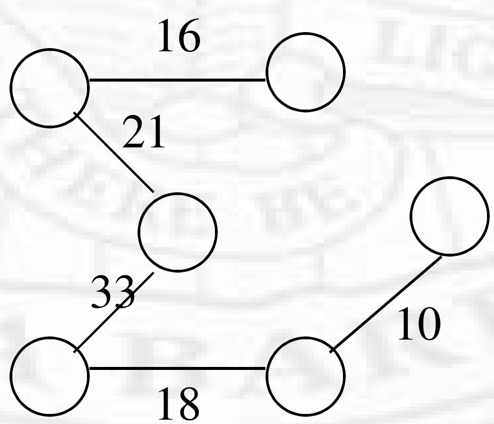
  } (Spanning tree)
  - the sum of labels (costs) of all tree branches is minimum among all spanning trees

- Objective function: $\sum_{i \in SP} cost(e_i)$
- Feasibility constraint: a tree containing all vertices
- Example




$$\sum_{i \in SP} cost(e_i) = 56$$



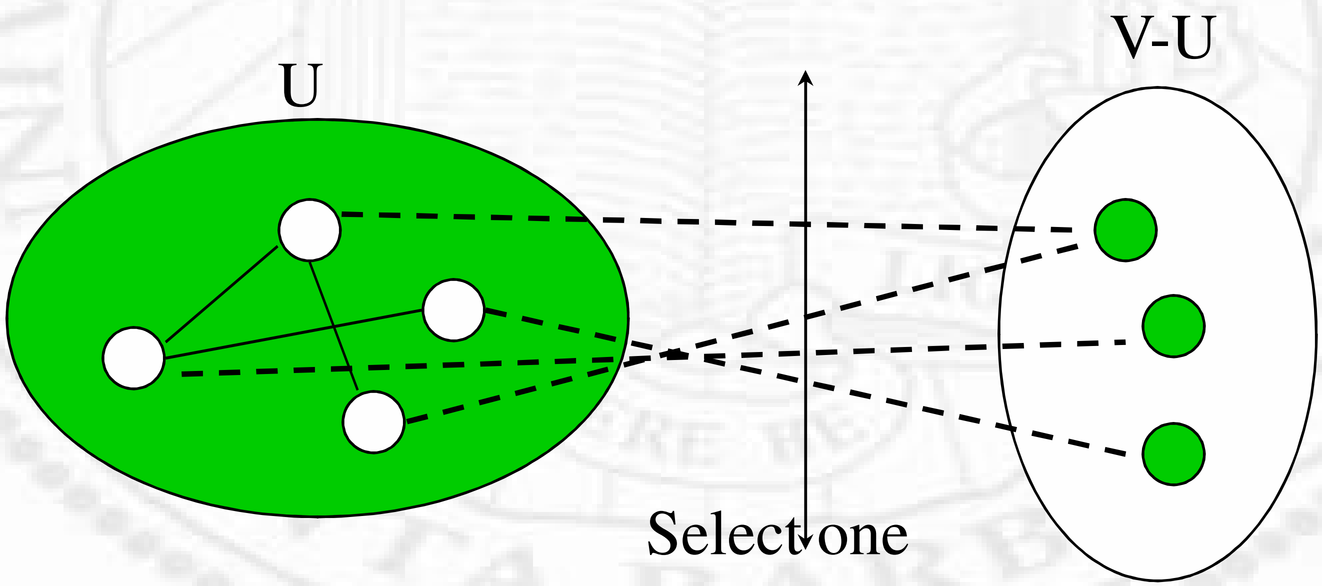




$$\sum_{i \in SP} cost(e_i) = 98$$

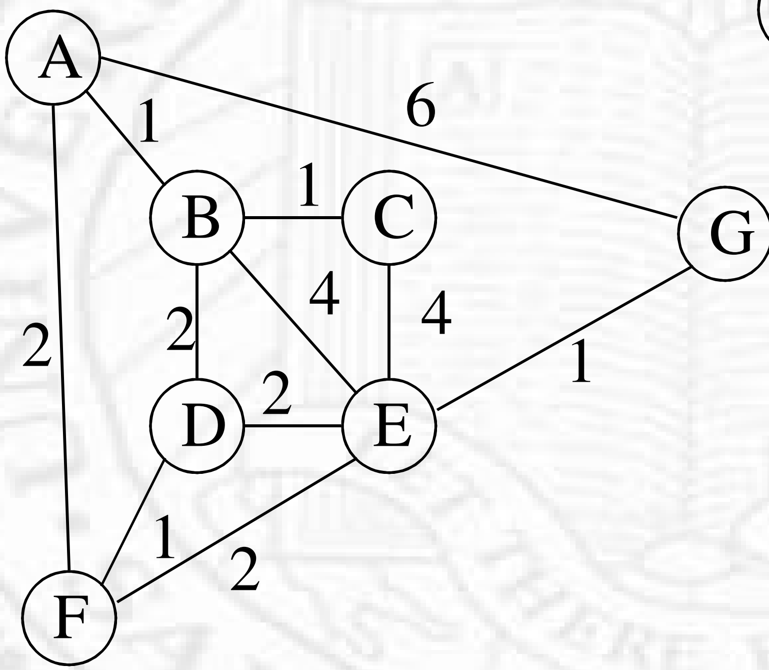
- *SELECT:* At each step, choose an edge with minimum cost (*optimality*) such that (*feasibility*):
  - the partial solution is always a tree (Prim)
  - the partial solution has potential of becoming a tree (no cycles, but not necessarily connected) (Kruskal)

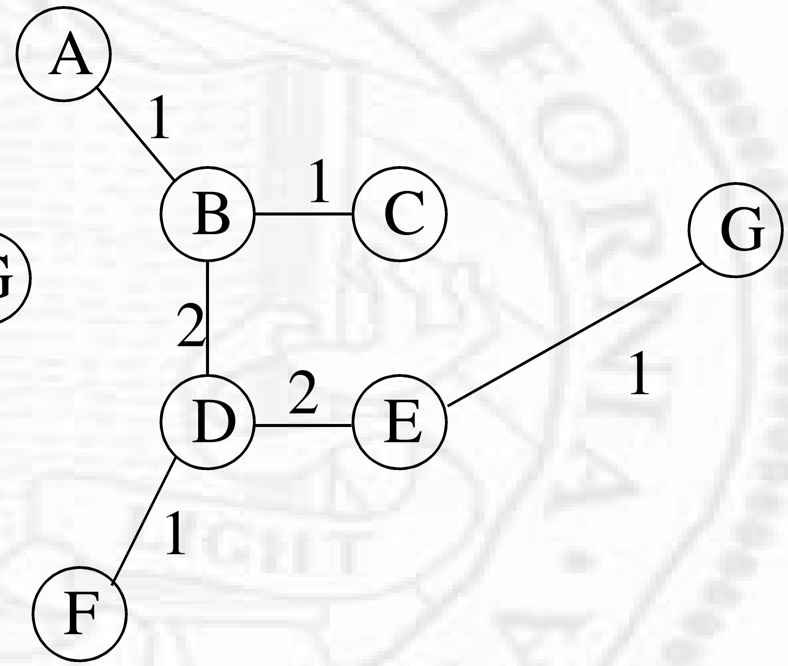
# *Prim's algorithm*

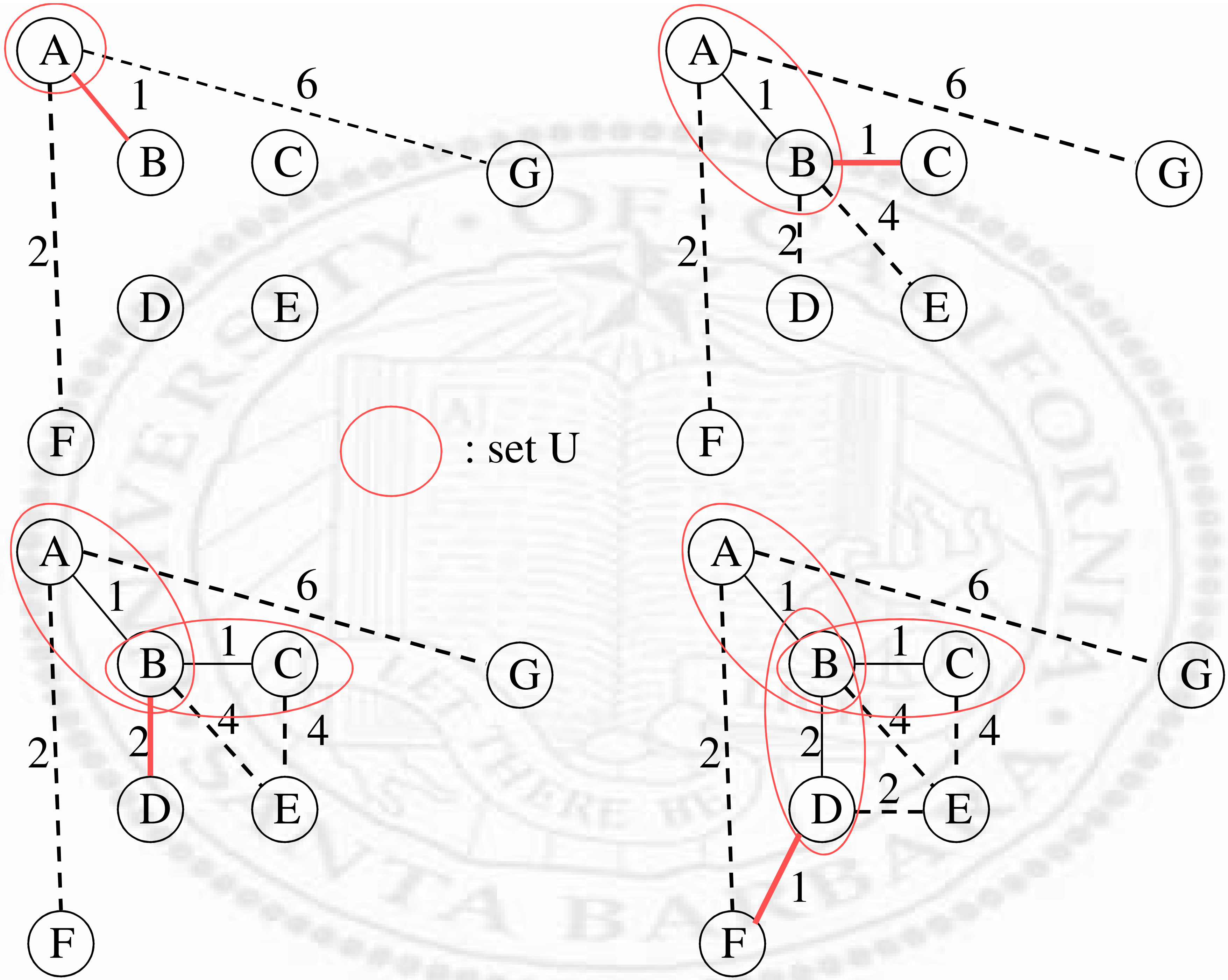
- First step: select a minimum cost edge, include it in the solution
- Other steps: select an edge *(u,v)*, *u* in *U* and *v* in *V-U*, until all vertices are counted for



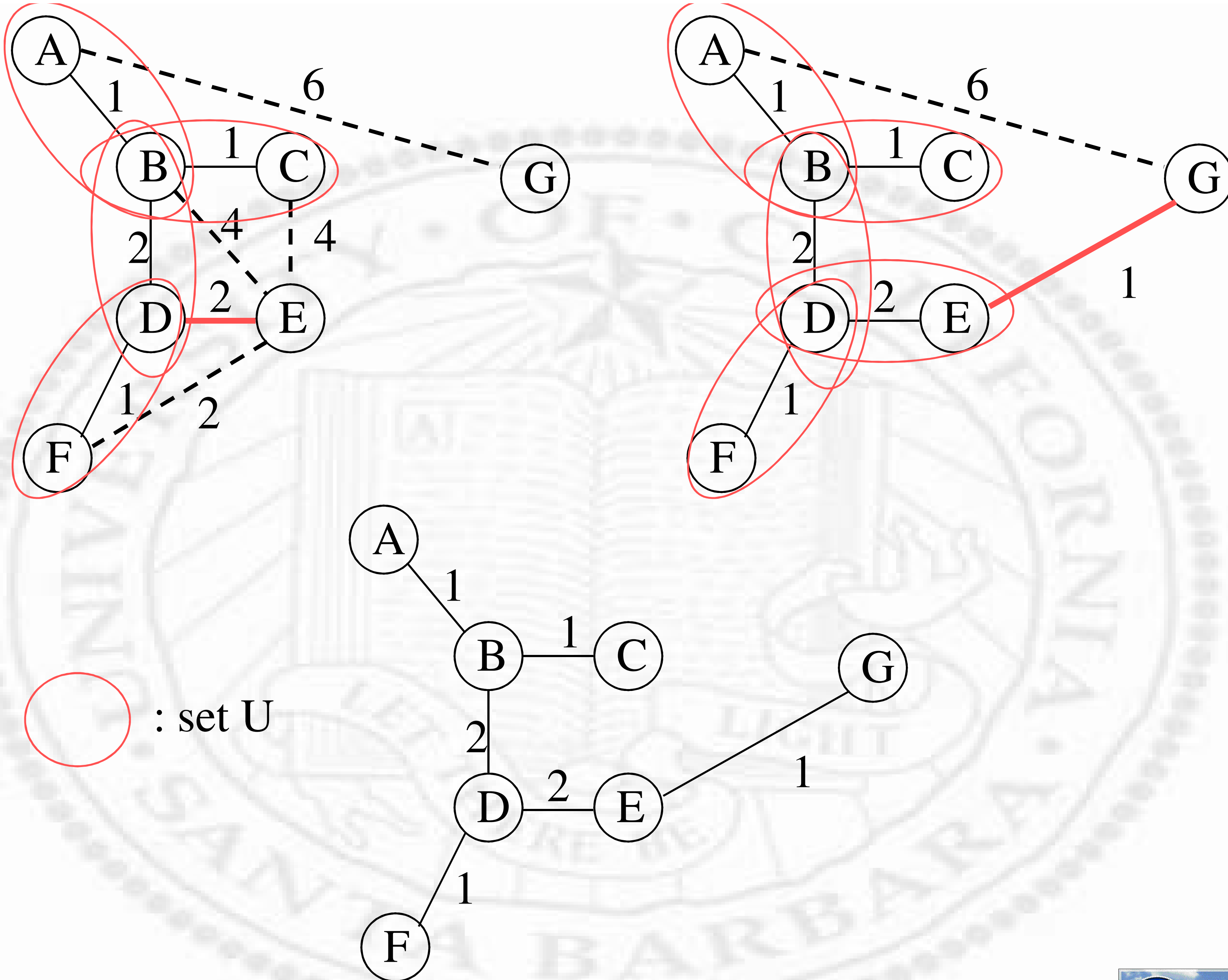
# *Example*





: set U

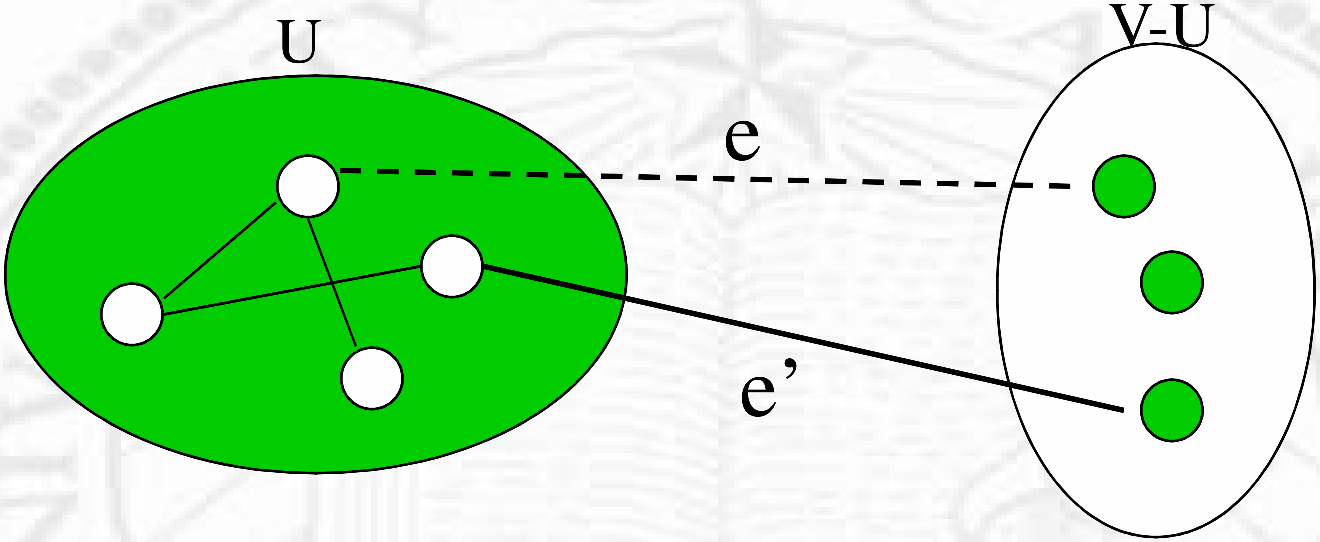

A
B
C
D
E
F
G
1
1
2
4
4
6
2
1
2
: set U

| Step | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ | $G$ |
|---|---|---|---|---|---|---|---|
| 1 | $-$ | $\underline{(1,A)}$ | $(\infty,A)$ | $(\infty,A)$ | $(\infty,A)$ | $(2,A)$ | $(6,A)$ |
| 2 | $-$ | $-$ | $\underline{(1,B)}$ | $(2,B)$ | $(4,B)$ | $(2,A)$ | $(6,A)$ |
| 3 | $-$ | $-$ | $-$ | $\underline{(2,B)}$ | $(4,B)$ | $(2,A)$ | $(6,A)$ |
| 4 | $-$ | $-$ | $-$ | $-$ | $(2,D)$ | $\underline{(1,D)}$ | $(6,A)$ |
| 5 | $-$ | $-$ | $-$ | $-$ | $\underline{(2,D)}$ | $-$ | $(6,A)$ |
| 6 | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $(1,E)$ |

$Cost_i = \min(Cost_i, Cost(new,i))$ Cost update

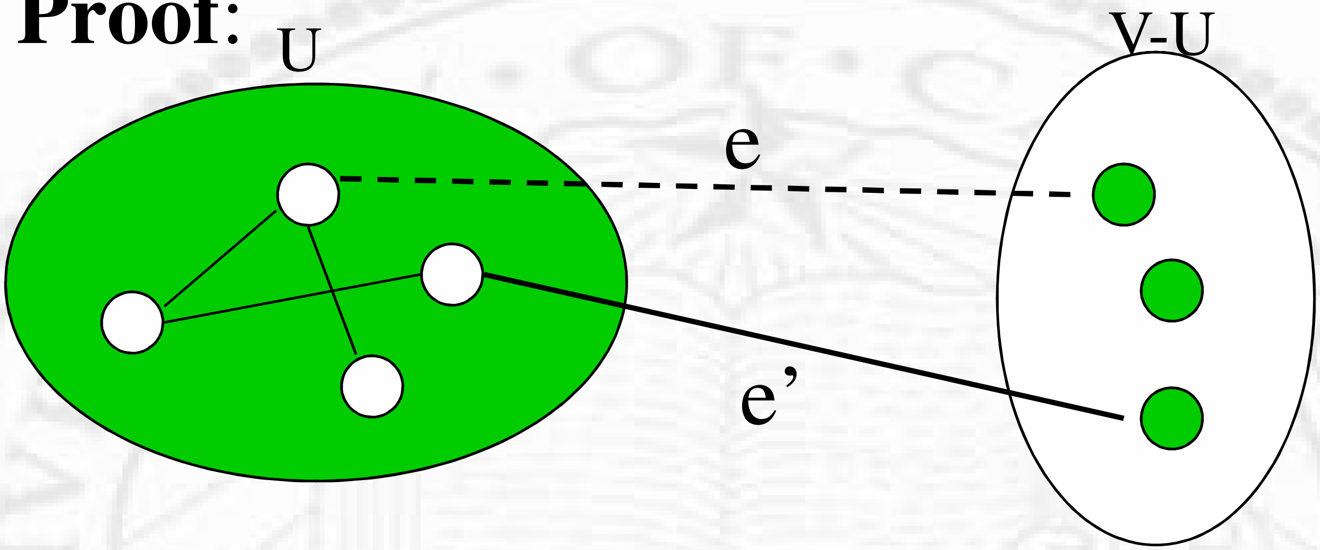
$Closest_i = (Cost_i == Cost(new,i))\,?\,new{:}Closet_i$ Nearest neighbor update

- **Proposition**: Prim’s algorithm finds MCST
- **Proof**:

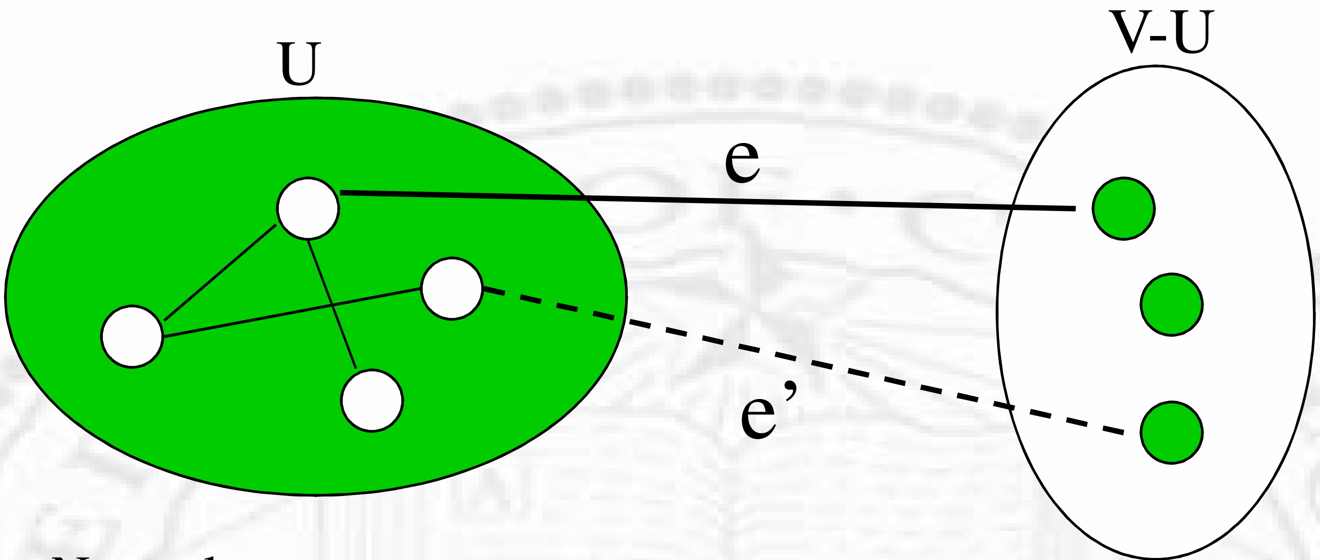



  - Again, there are two solutions, PRIM and MCST
  - They better differ, and MCST has a lower cost
  - In the construction of PRIM, if an edge e is considered
    - It is in MCST, ok, continue (cannot be forever)
    - If it is not in MCST, then ….

- **Proposition**: Prim’s algorithm finds MCST
- **Proof**:




  - Let U be the subgraph (tree) considered so far
  - Let V-U be the remaining part, then
  - There must be at least one edge (e’) chosen between U and V-U in MCST
    - Prim’s algorithm selects the minimum cost one (*e*)
    - *e’* can be replaced by *e* in the MCST




- No cycle
  - U has no cycle
  - V-U has no cycle
  - Between U and V-U cannot has cycle w. a single path e
- Still connected
  - U is connected
  - V-U is connected
  - U and V-U connected through either e or e’
- The same number of edges => it is a spanning tree
- A tree of a smaller cost

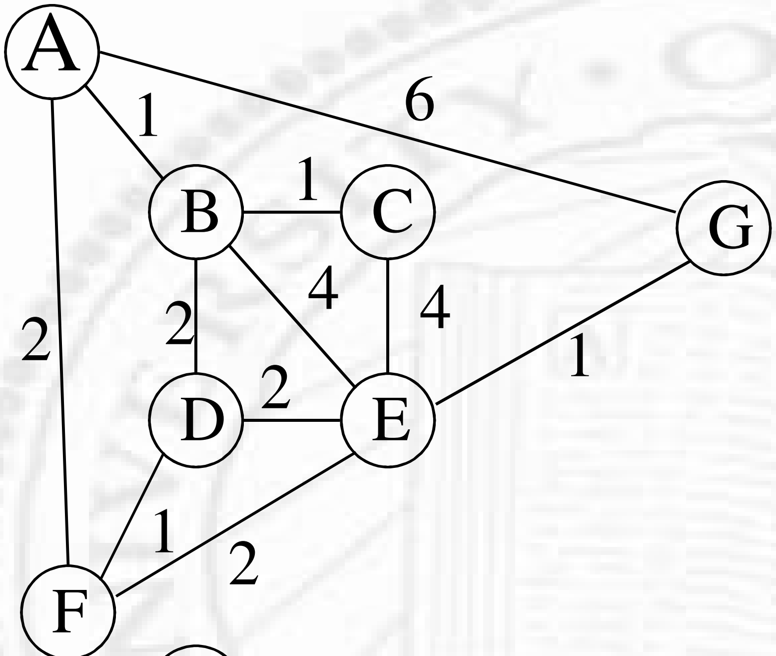
- Time complexity
  - Totally *n* vertices have to be connected
  - Each time an edge is added, one additional vertex is accounted for
    - Loop through *n-1* times
  - Through each loop

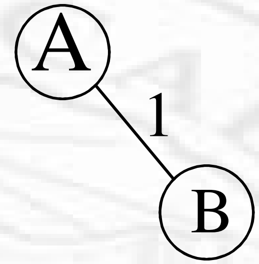
$O(n\text{-}i)$ — Select the edge of a minimum cost from *U to V-U*

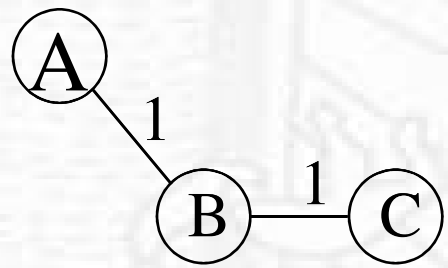
$O(n\text{-}i)$ — Update the nearest vertex and cost for vertices in *V-U'*

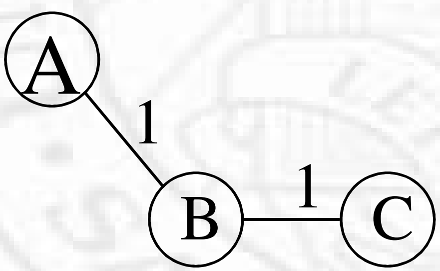
$$\sum_{i=1}^{n-1}(n-i)=O(n^2)$$

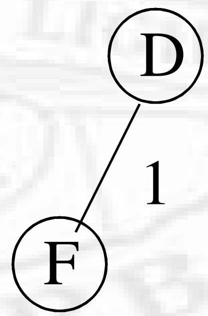
# *Kruskal's algorithm*



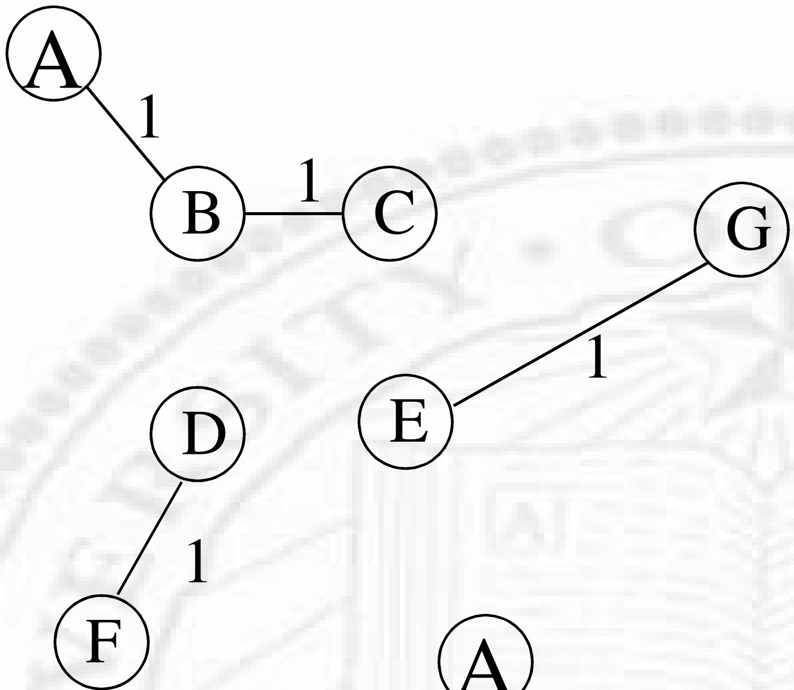



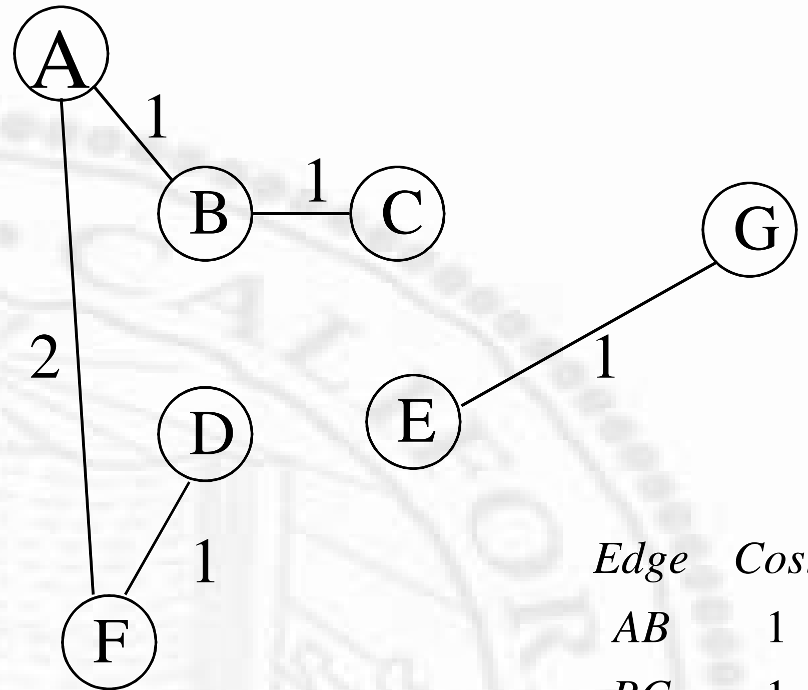



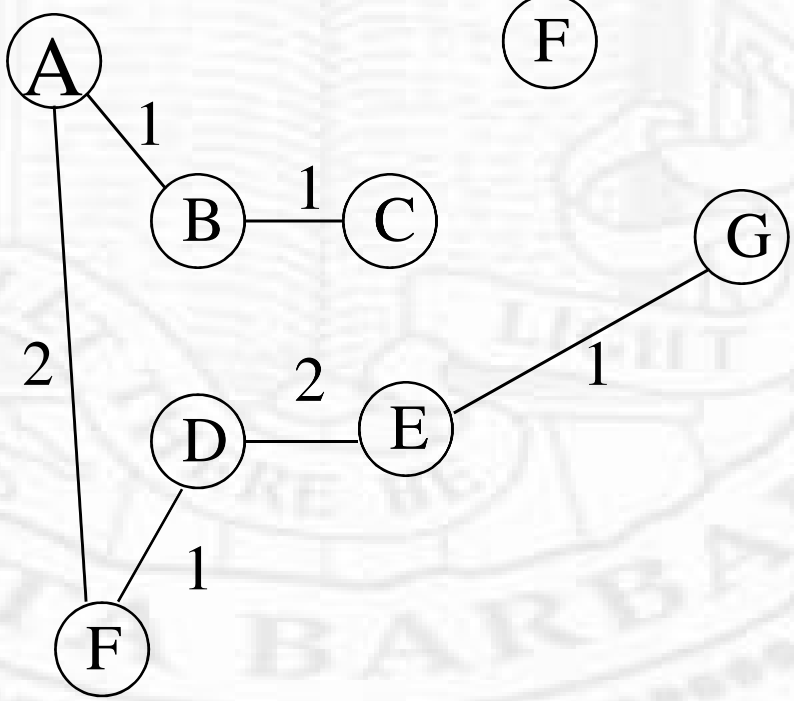




| *Edge* | *Cost* |
|---|---|
| *AB* | 1 |
| *BC* | 1 |
| *DF* | 1 |
| *EG* | 1 |
| *AF* | 2 |
| *BD* | 2(×) |
| *DE* | 2 |
| *EF* | 2(×) |
| *BE* | 4(×) |
| *CE* | 4(×) |
| *AG* | 6(×) |




| *Edge* | *Cost* |
|---|---|
| *AB* | 1 |
| *BC* | 1 |
| *DF* | 1 |
| *EG* | 1 |
| *AF* | 2 |
| *BD* | 2(×) |
| *DE* | 2 |
| *EF* | 2(×) |
| *BE* | 4(×) |
| *CE* | 4(×) |
| *AG* | 6(×) |

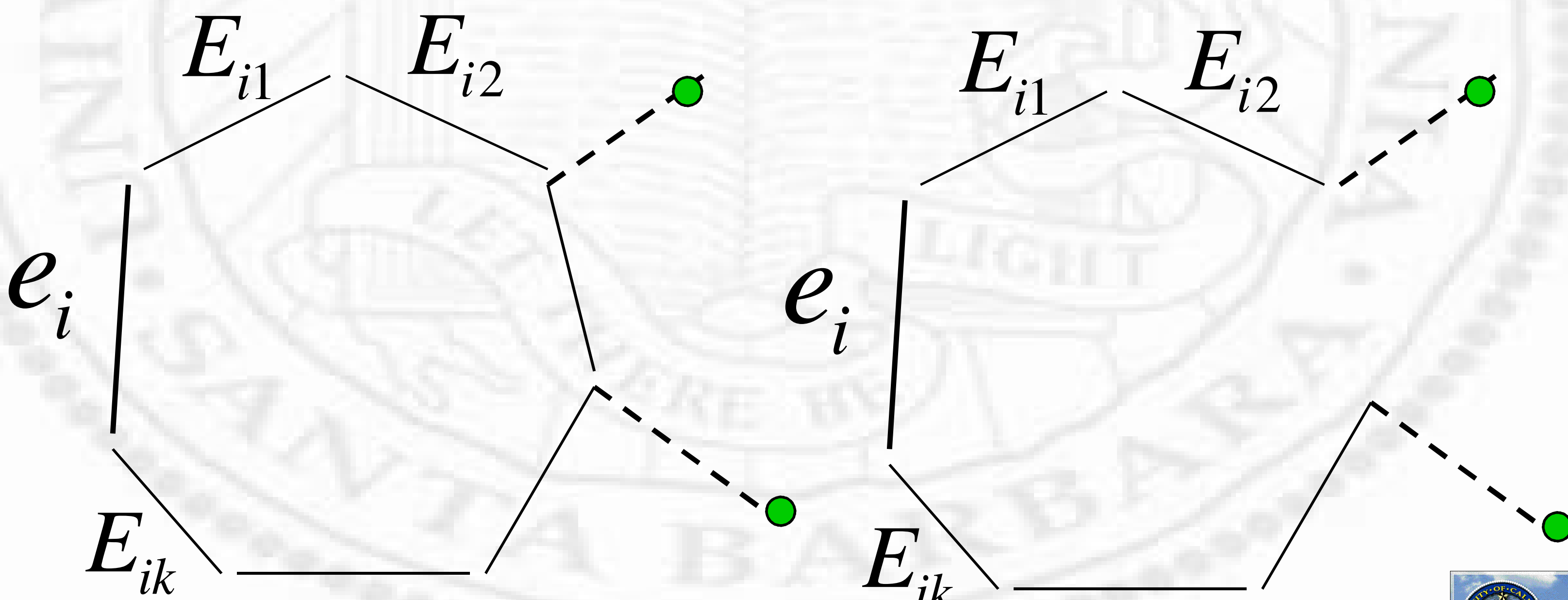
- **Proposition**: Kruskal's algorithm find MCST
- **Proof**:

$$
\begin{array}{cc}
T & T' \\
Kruskal's & MST \\
e_1 & E_1 \\
e_2 & E_2 \\
e_3 & E_3 \\
\vdots & \vdots \\
e_i & E_i \longleftarrow \\
\vdots & \vdots \\
e_e & E_e
\end{array}
$$

first index the two solutions differ

$$Cost\ (e_i) \le Cost\ (E_j) \quad i \le j \le e$$

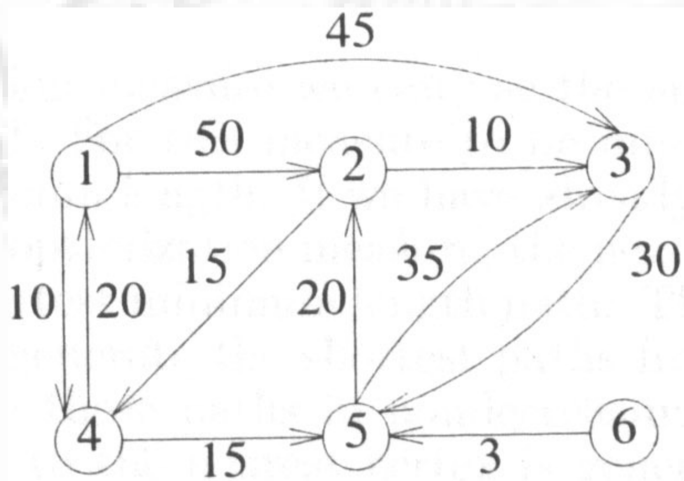
- Including $e_i$ in MCST creates a cycle
  - Not all edges in the cycle belongs to T (Kruskal's)
  - At least one of them must have a higher costs
  - Remove that high-cost edge breaks the cycle and maintain the tree structure



- Time complexity
  - Total *e* edges are considered in order of nondecreasing cost
    - Use partially-ordered tree (heap) to represent edges
      - Construction $O(e \log e)$
      - Deletemin $O(e \log e)$
  - At each step, remove edge with a minimum cost and check to see whether it creates a cycle if included
    - Use Union-and-Find tree
      - Initially each vertex in a set by itself
      - Inclusion of an edge, join the sets containing the edge's two end points
      - Edges are not included if the two end points are in the same set
  - $O(e \log e)$

# *Single-Source Shortest Path*

- Input:
  - *G=(V,E),* an directed, labeled graph
  - A source vertex
- Output:
  - The shortest path, from source to every other vertices in the graph, if one exists



(a) Graph

| | *Path* | *Length* |
|---|---|---|
| 1) | 1, 4 | 10 |
| 2) | 1, 4, 5 | 25 |
| 3) | 1, 4, 5, 2 | 45 |
| 4) | 1, 3 | 45 |

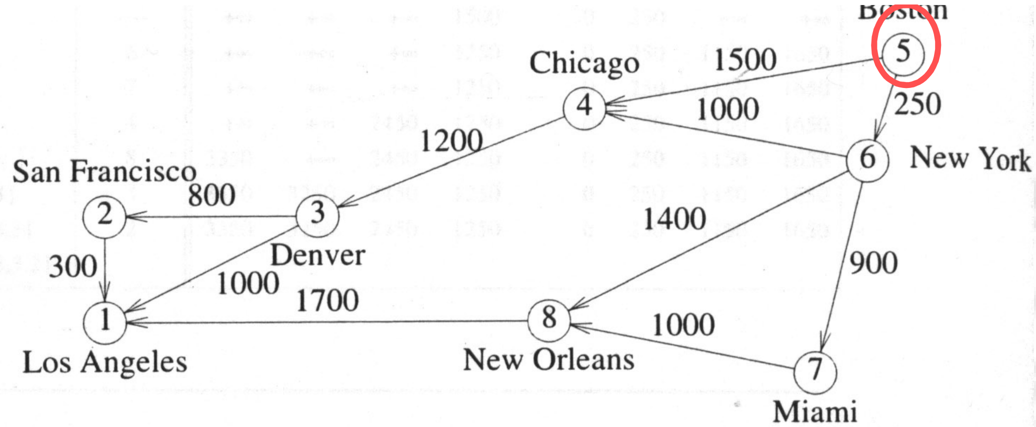
(b) Shortest paths from 1

# *Possible Greedy Strategies*

- Exploring a maze where you cannot see beyond the first turn
- Extremely greedy: with no memory, go where the path leads you (good paths can turn bad at any instance)
- Cautiously greedy: with memory, go where the shortest path encountered so far (backtracking to the path necessary)

# *Greedy Selection*

1. Visited set = {$s$}
2. From visited set, find all 1-distance (direct edge) neighbors
3. Visit the one with the shortest distance: $n$
4. Enlarge visited set = visited set U {$n$}
5. Update distances to the remaining vertices
    1. Go through original visited set
    2. Go through $n$
6. Go back to 2



(a) Digraph

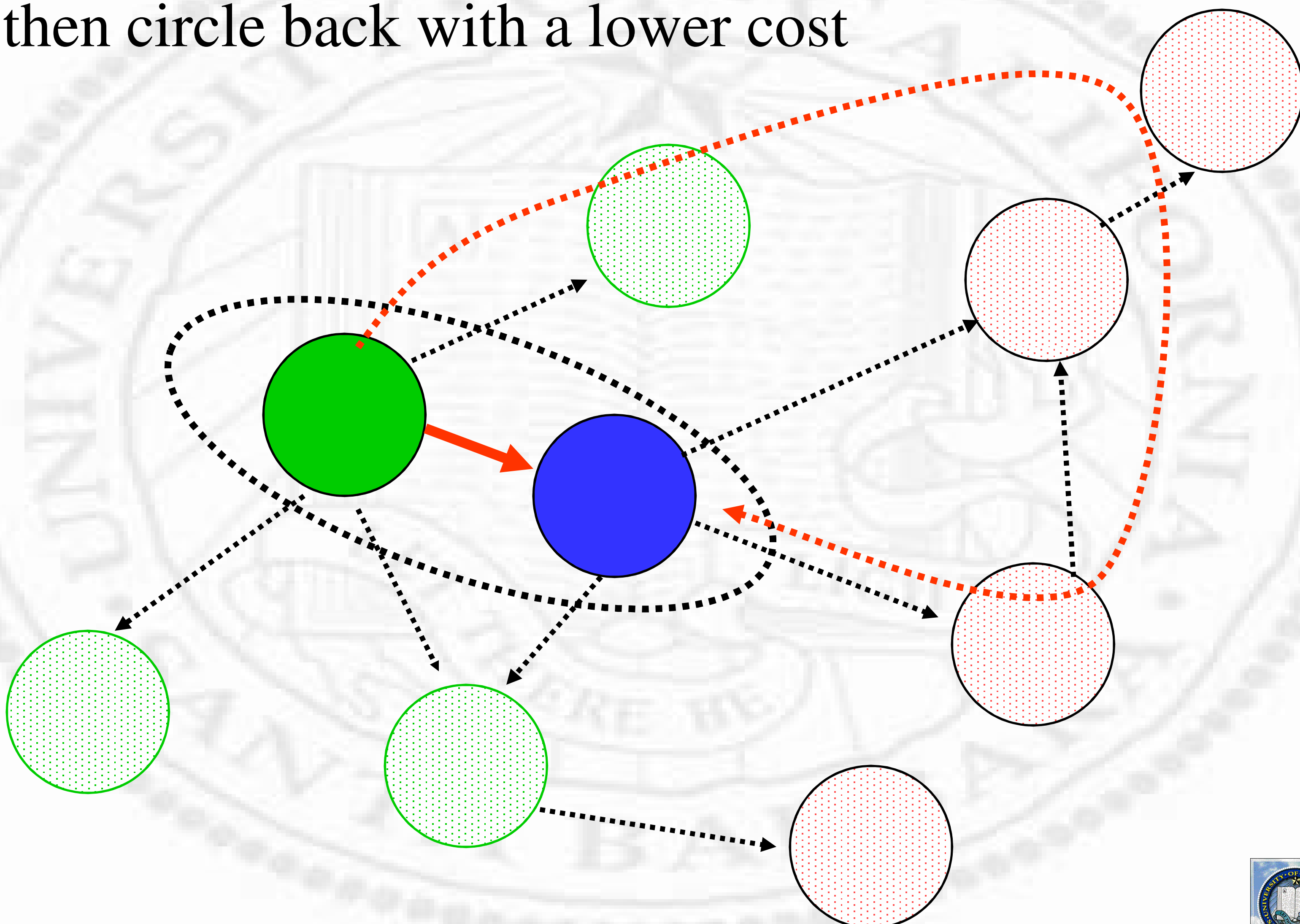
```
If (dist(w)>dist(n)+cost(n,w)) {
    dist(w) = dist(n)+cost(n,w);
    previous_neighbor = n;
}
```

| Iteration | *S* | Vertex selected | Distance | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | LA [1] | SF [2] | DEN [3] | CHI [4] | BOST [5] | NY [6] | MIA [7] | NO [8] |
| Initial | -- | ---- | +∞ | +∞ | +∞ | 1500 | 0 | 250 | +∞ | +∞ |
| 1 | {5} | 6 | +∞ | +∞ | +∞ | 1250 | 0 | 250 | 1150 | 1650 |
| 2 | {5,6} | 7 | +∞ | +∞ | +∞ | 1250 | 0 | 250 | 1150 | 1650 |
| 3 | {5,6,7} | 4 | +∞ | +∞ | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 4 | {5,6,7,4} | 8 | 3350 | +∞ | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 5 | {5,6,7,4,8} | 3 | 3350 | 3250 | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 6 | {5,6,7,4,8,3} | 2 | 3350 | 3250 | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| | {5,6,7,4,8,3,2} | | | | | | | | | |

Complexity: $O(n^2)$

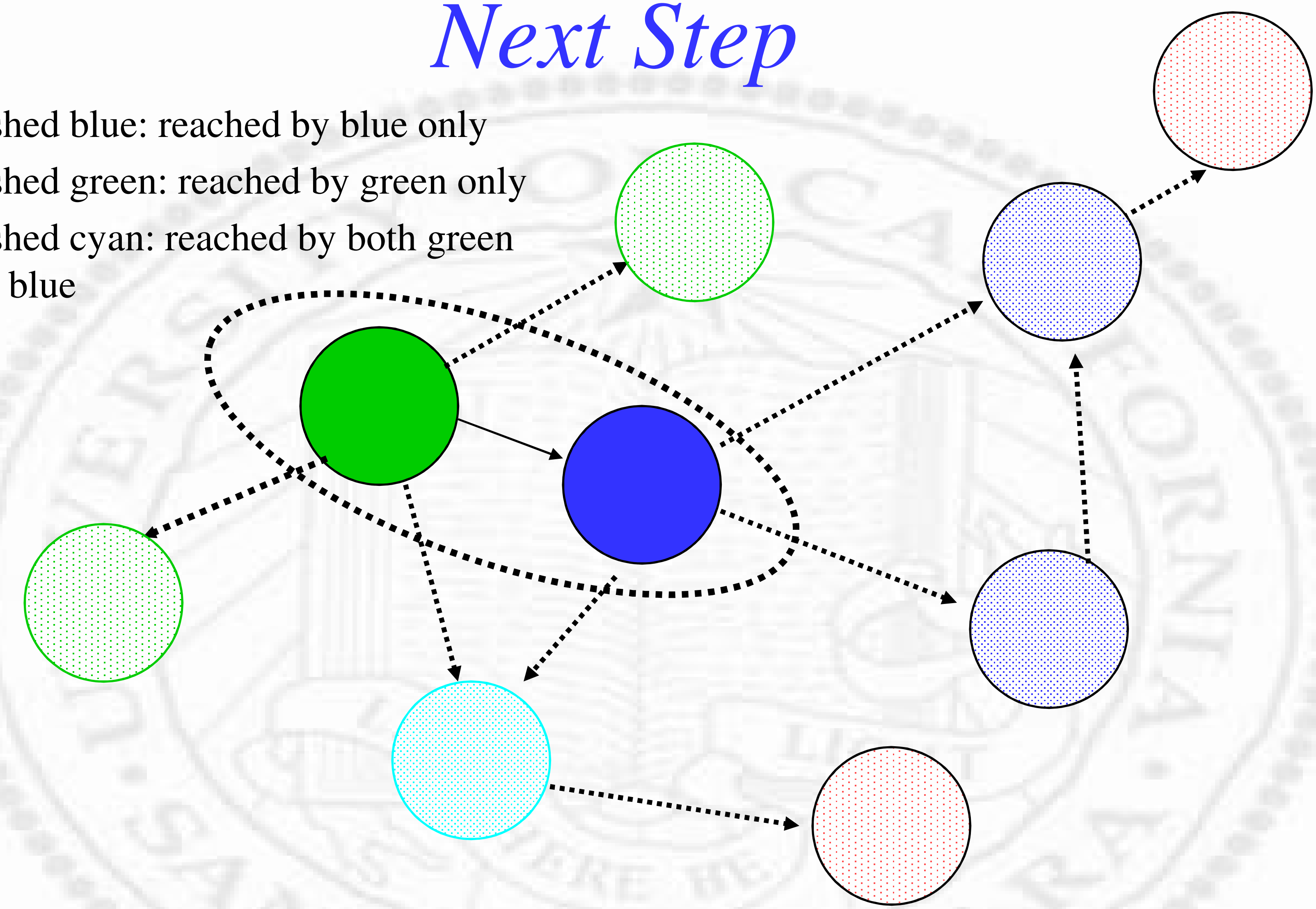
# *Initially*

- You cannot go to blue through dashed green and then circle back with a lower cost



# *Next Step*

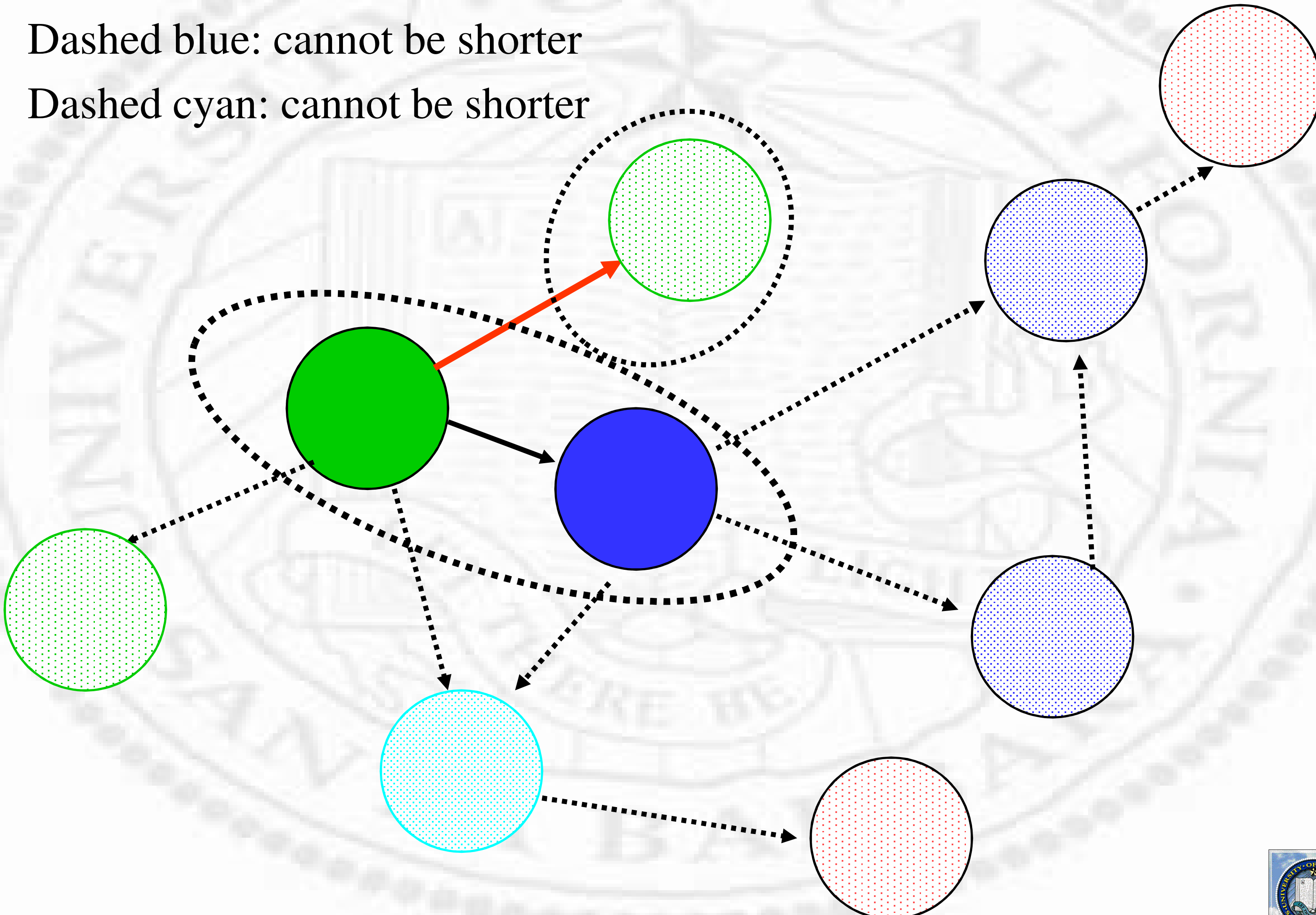
- Dashed blue: reached by blue only
- Dashed green: reached by green only
- Dashed cyan: reached by both green and blue



- One of the dashed blue, green, or cyan will be visited next (i.e., the shortest path to the visited node is determined greedily)
- Is that possible to go through other dashed blue, green, or cyan and circle back to the visited node with a shorter path?

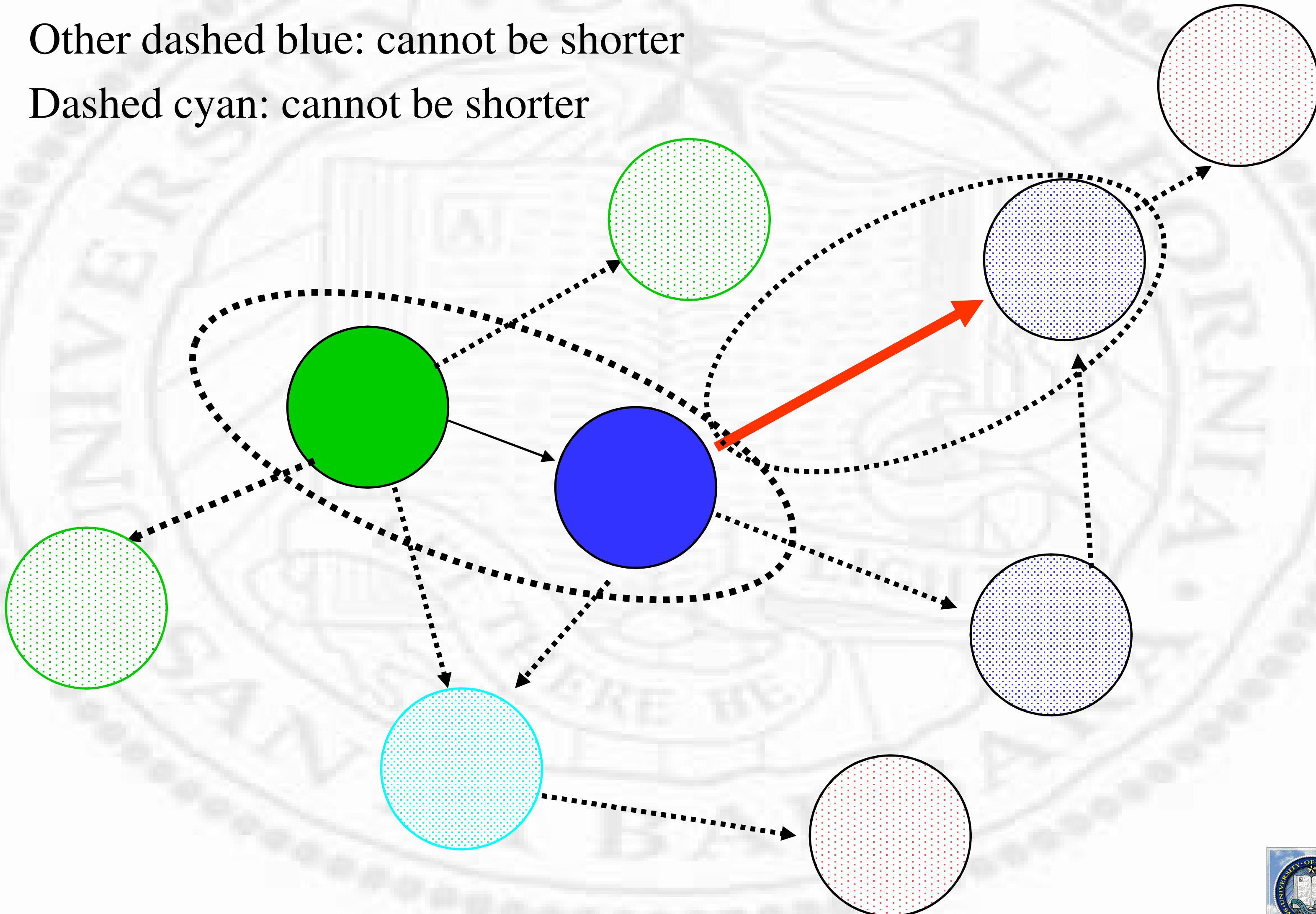
# *Case one: Dashed green is selected*

- Other dashed green: cannot be shorter
- Dashed blue: cannot be shorter
- Dashed cyan: cannot be shorter



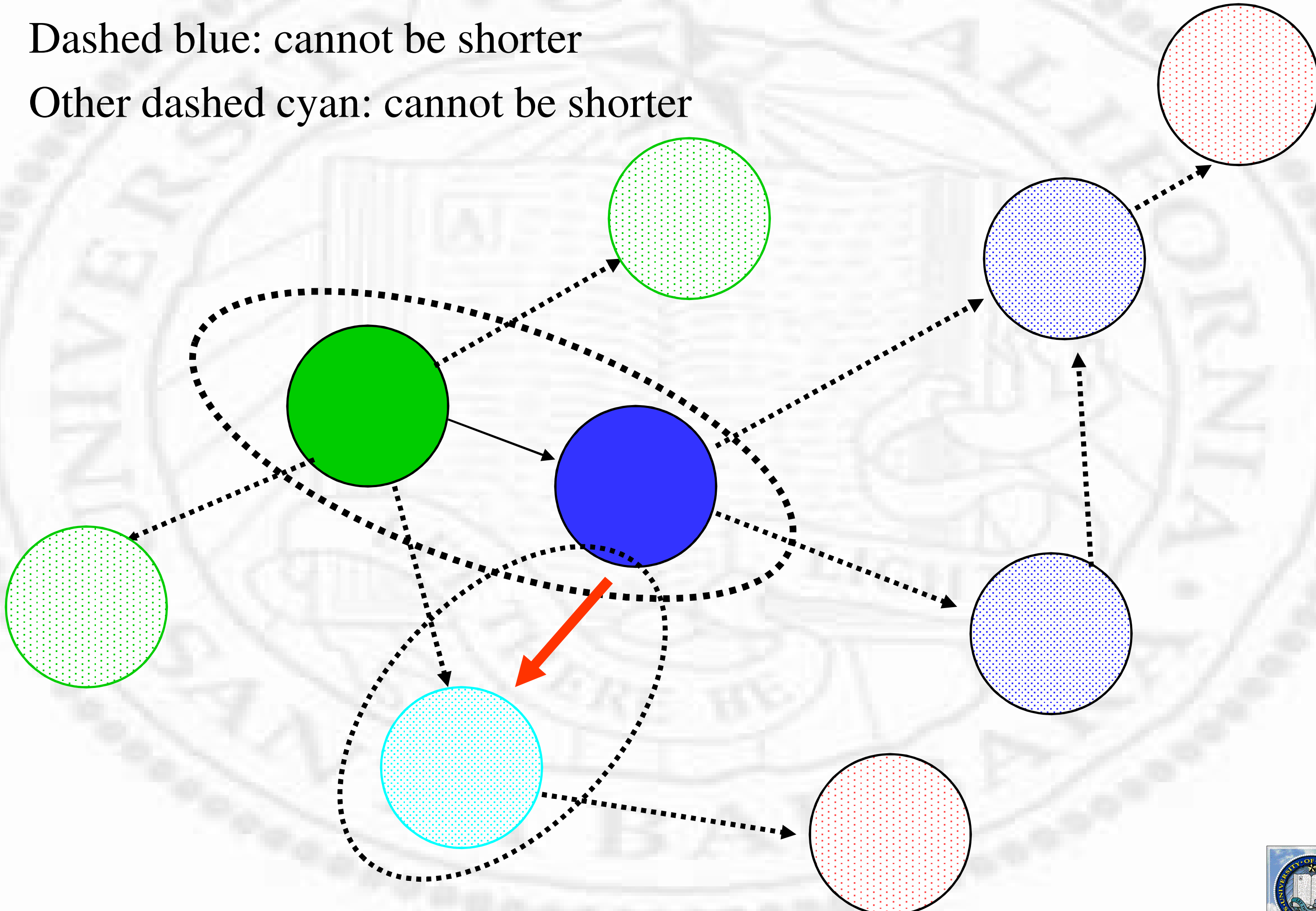
# *Case two: Dashed blue is selected*

- Dashed green: cannot be shorter
- Other dashed blue: cannot be shorter
- Dashed cyan: cannot be shorter

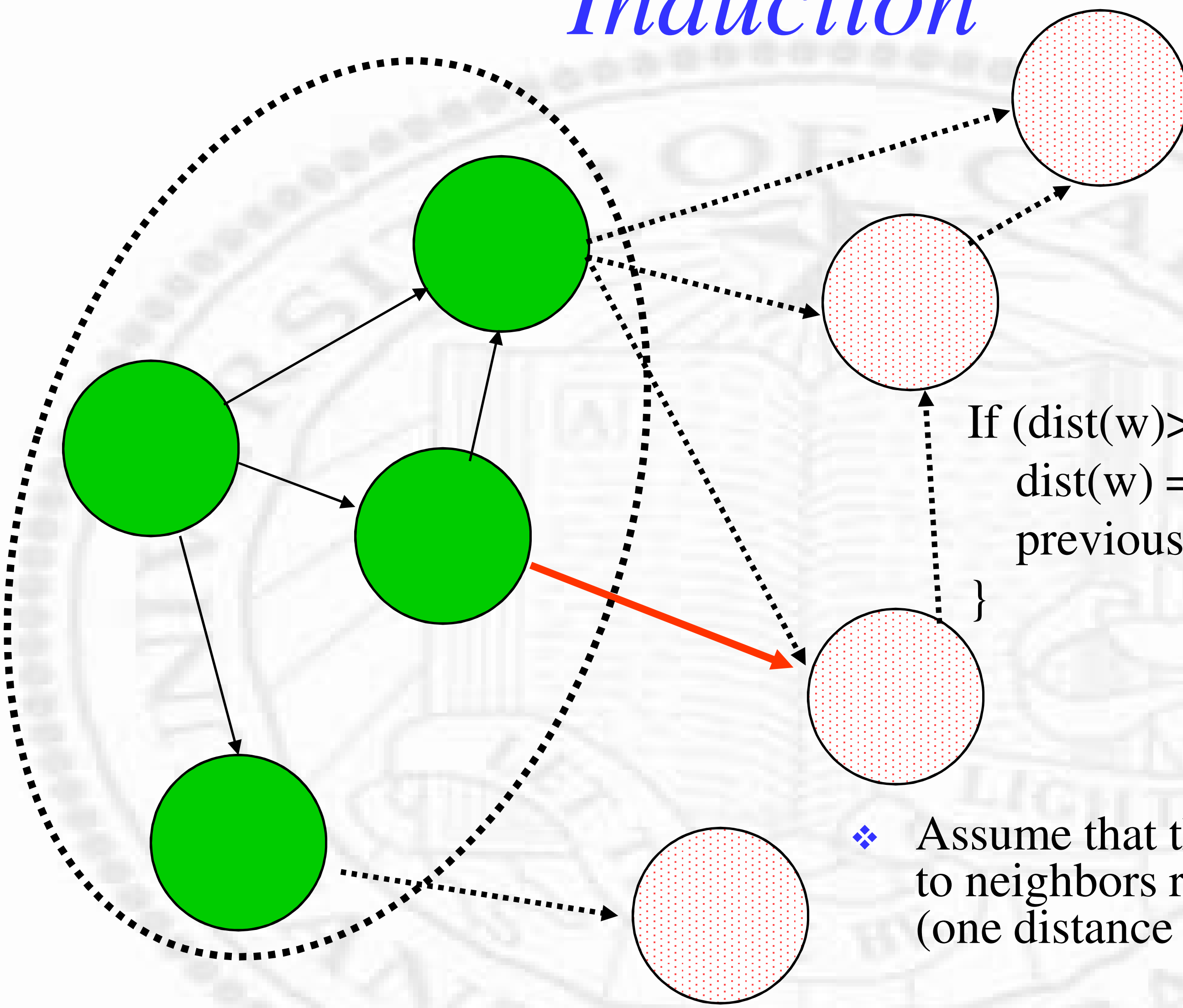


# *Case three: Dashed cyan is selected*

- Dashed green: cannot be shorter
- Dashed blue: cannot be shorter
- Other dashed cyan: cannot be shorter



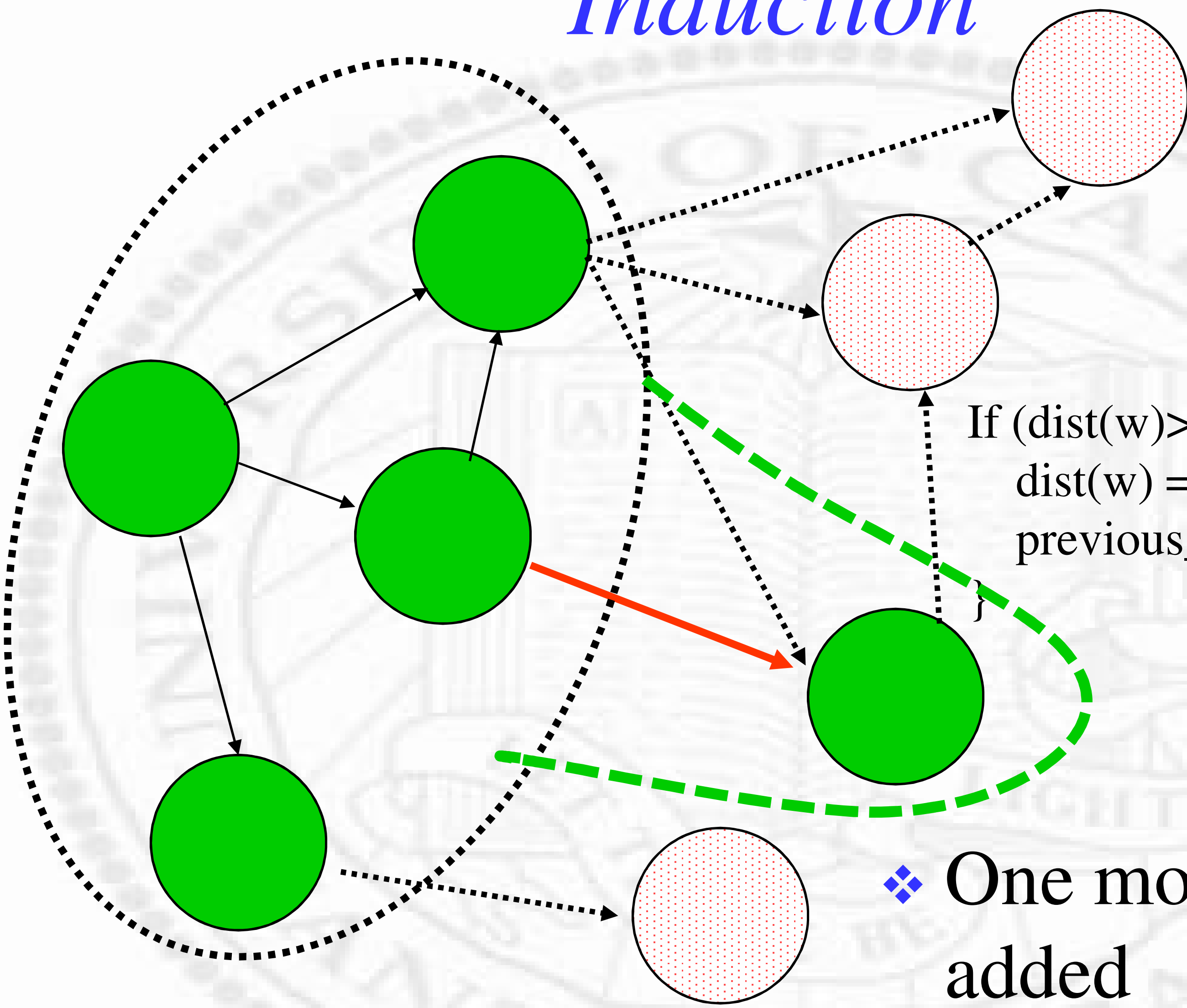
# *Induction*



If (dist(w)>dist(n)+cost(n,w)) {
    dist(w) = dist(n)+cost(n,w);
    previous_neighbor = n;
}

- Assume that the (current) shortest path to neighbors right outside the wall (one distance away) has been found

# *Induction*



```
If (dist(w)>dist(n)+cost(n,w)) {
    dist(w) = dist(n)+cost(n,w);
    previous_neighbor = n;
}
```

- One more node is added

- Three things can happen for a node still outside the wall (the envelop) after a new node is added
  - Not reached by the new node
    - The current best path didn't change
  - Reached by the new node but not any node in the previous envelop
    - The current best path must be the one via the new node
  - Reached by the new node and also nodes in the previous envelop
    - The update process should record the best between the two
- Hence, when "the best of the best" is chosen to go out the wall, one cannot jump through other paths on the wall and circle back to get a better result

# *Job Sequencing with Deadlines*

- Input:
  - a set of $n$ jobs, each with a deadline and a profit if completed before deadline
  - one machine to execute all the jobs
  - each job takes one unit of time
- Output:
  - a subset of jobs, each completed before deadline, with maximum profit

- Objective function: $\max \sum_{i \in J} P_i$
- Feasibility constraint:
- Example:

$n = 4, (P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$

$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1$

| feasible | schedule | profit |
|---|---|---|
| (1,2) | 2,1 | 110 |
| (1,3) | 1,3 or 3,1 | 115 |
| (1,4) | 4,1 | 127 |
| (2,3) | 2,3 | 25 |
| (3,4) | 4,3 | 42 |
| (1) | 1 | 100 |
| (2) | 2 | 10 |
| (3) | 3 | 15 |
| (4) | 4 | 27 |

- *SELECT:* select the job with maximum profit subject to the constraint that the resulting schedule is still feasible

| | $J$ | $\sum_{i \in J} P_i$ | |
|---|---|---|---|
| *Initially* | $\Phi$ | 0 | |
| 1 | (1) | 100 | |
| 4 | (1,4) | 127 | |
| 3 | (1,4) | 127 | (1,4,3) *not feasible* |
| 2 | (1,4) | 127 | (1,4,2) *not feasible* |

(Q1) How to determine if *J* is feasible?

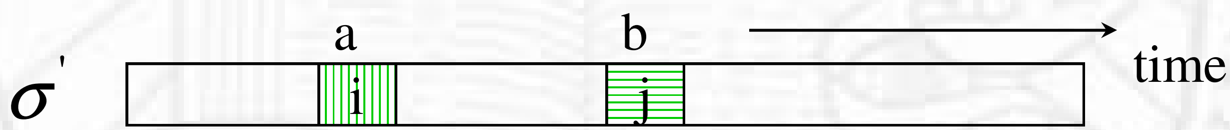
(Q2) Is greedy algorithm optimal?

(Q1) If *J={1,2,3,…,k}*

- try all possible (*k!*) permutations (schedules) and see whether at least one of them allows all jobs to be finished before their deadlines
- intuitively, jobs with earlier deadline (more urgent) should be performed first
- check the permutation $\sigma^* = (i_1, i_2, \ldots, i_k)$

$$d_{i_1} \le d_{i_2} \le \ldots \le d_{i_k}$$

- **Proposition**: $J=\{1,2,\ldots,k\}$ is feasible if and only if $\sigma^*$ is feasible
- **Proof**:
  - If $\sigma^*$ is feasible, then $J=\{1,2,\ldots,k\}$ is feasible (by definition)
  - If $J=\{1,2,\ldots,k\}$ is feasible, then




$d_i \geq a$ job completed before deadline

$d_j \geq b$

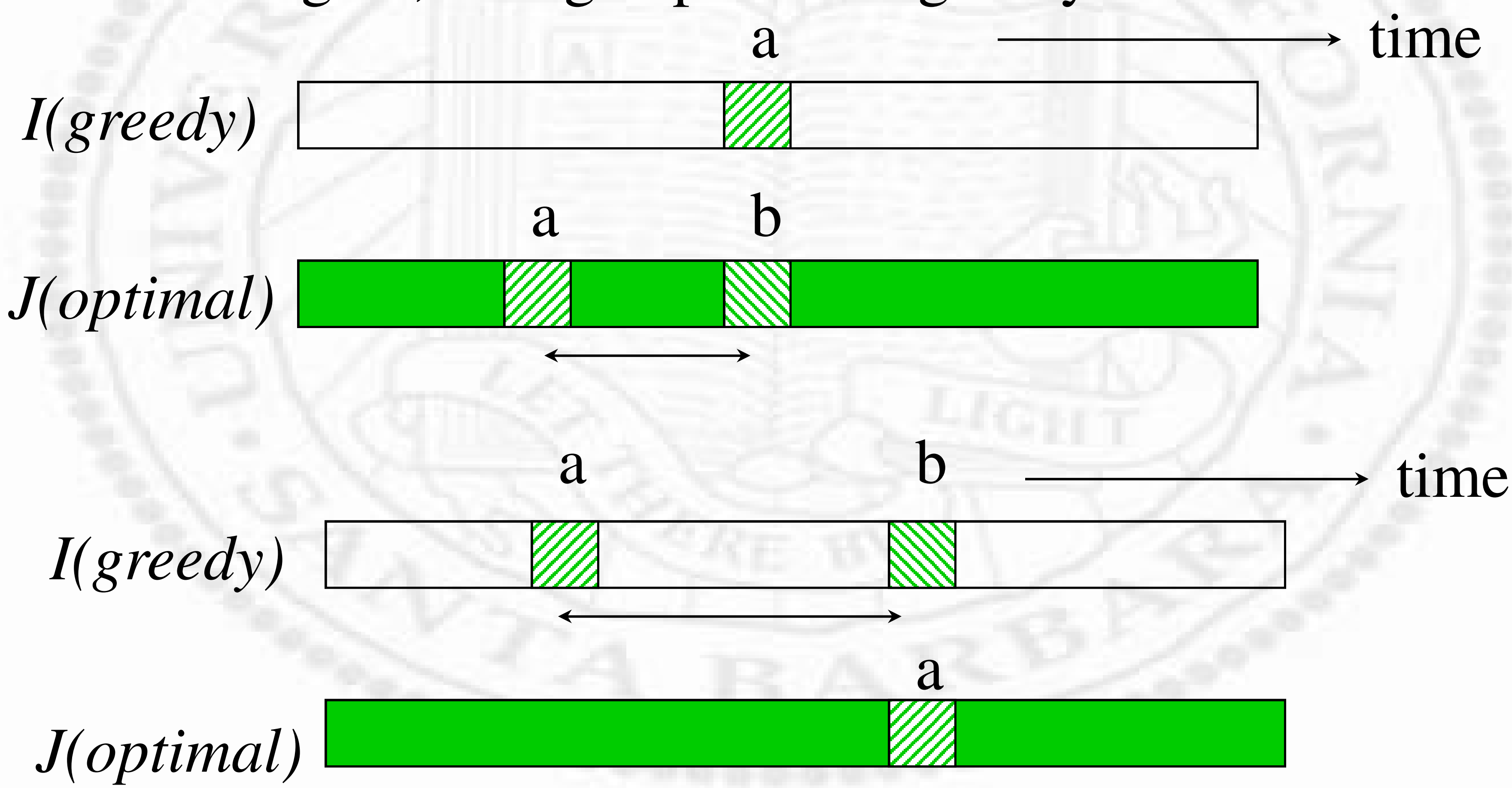
$d_i \geq d_j$ out of order

$d_j \geq b > a$ j can be moved forward

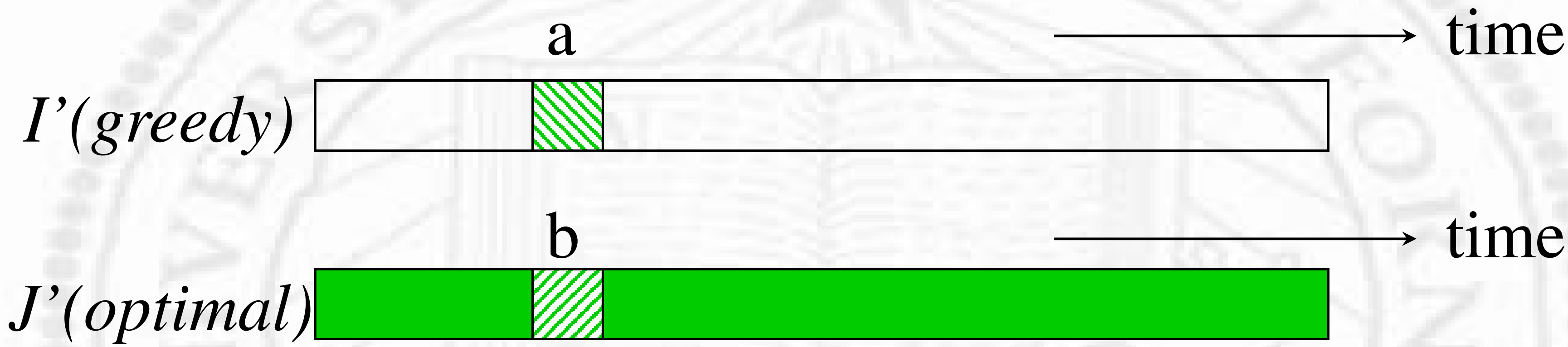
$d_i \geq d_j \geq b$ i can be moved backward

- **Proposition**: The greedy method produces a schedule with the maximum profit
- **Proof**:
  - Two *different* solutions: optimal and greedy
  - Jobs that are in both optimal and greedy
    - make sure that they are scheduled at the same time
  - Jobs that are in one but not the other
    - change them into ones in greedy without decreasing profit
  - The process continues until two solutions are equal

- For jobs that are in both
  - the job is scheduled the same in both
  - the job is scheduled earlier in optimal
  - the job is scheduled earlier in greedy
  - Again, change optimal to greedy



- For jobs that are different
  - *I'* and *J'* are such that jobs common to both are scheduled at the same slot




$P_a \geq P_b \because$ if b has a larger profit and is feasible, it will appear in the greedy solution

  – Replace *b* with *a* in the optimal solution will not decrease the profit

# *Finally*

- Can it be that greedy solution still does more jobs than optimal?
  - No, optimal will not be optimal then
- Can it be that optimal solution does more jobs than greedy?
  - No, if such a job is feasible, how come greedy solution doesn't include it?

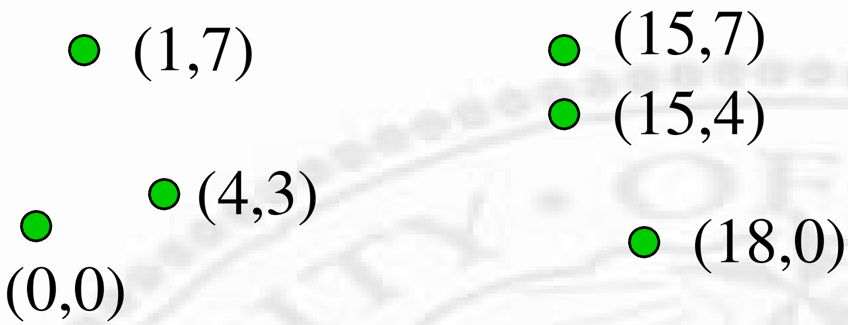
- Time complexity
  - Sort jobs according to nondecreasing profit $O(nlogn)$
  - Consider $n$ jobs in turn
    - for each job, insert the job into the partial solution using its deadline $O(i)$
    - check whether the new solution is still feasible $O(i)$

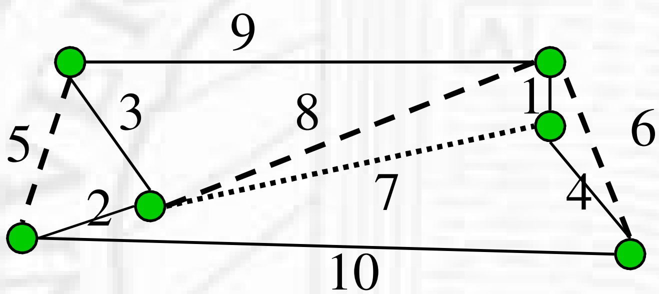
$$O(n^2)$$

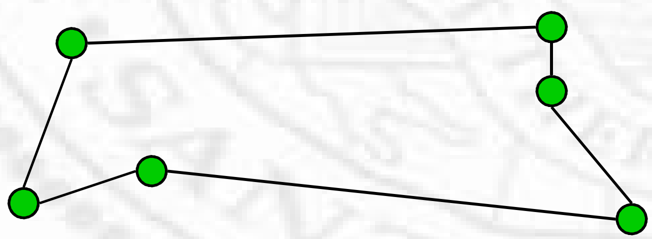
# *Greedy Method as Heuristics*

- For problems whose solutions are found by "try-all-possibilities," an optimal solution is difficult to compute for large problem size
- Greedy method can usually produce a "very good" solution at a fraction of the cost

- Example: Traveling salesperson's problem
  - Input: a fully connected, labeled undirected graph
  - Output: a tour (a simple cycle including all vertices) whose edge weights are minimum.
  - Greedy method:
    - A variant of Kruskal's algorithm
    - Consider edges in nondecreasing cost
    - The edge under consideration, together with all edges already selected:
      - do not cause a vertex to have a degree of three or more
      - do not form a cycle, unless the number of edges equals to that of vertices



- Greedy solution
  - 5,6 rejected: cycle
  - 7,8 rejected: vertex degree larger than 2
  - cost = 49.73
- Optimal solution
  - cost = 48.39