알고리즘

03. Greedy Algorithm

2014/11/03

미디어소프트웨어학과
민경하

Contents

3. Greedy algorithm

3.0 Basics

3.1 Minimum spanning trees

3.2 Optimal storage on tapes

3.3 Knapsack problem

3.4 Job sequencing with deadline

3.5 Optimal merge patterns

3.6 Huffman encoding

3.0 Basics

- Greedy algorithm
  - The most straightforward design technique to solve a problem
    - n inputs
    - Requires to obtain a subset that satisfies some constraints
  - Terms
    - Constraint
    - Objective function
    - Feasible solution
    - Optimal solution

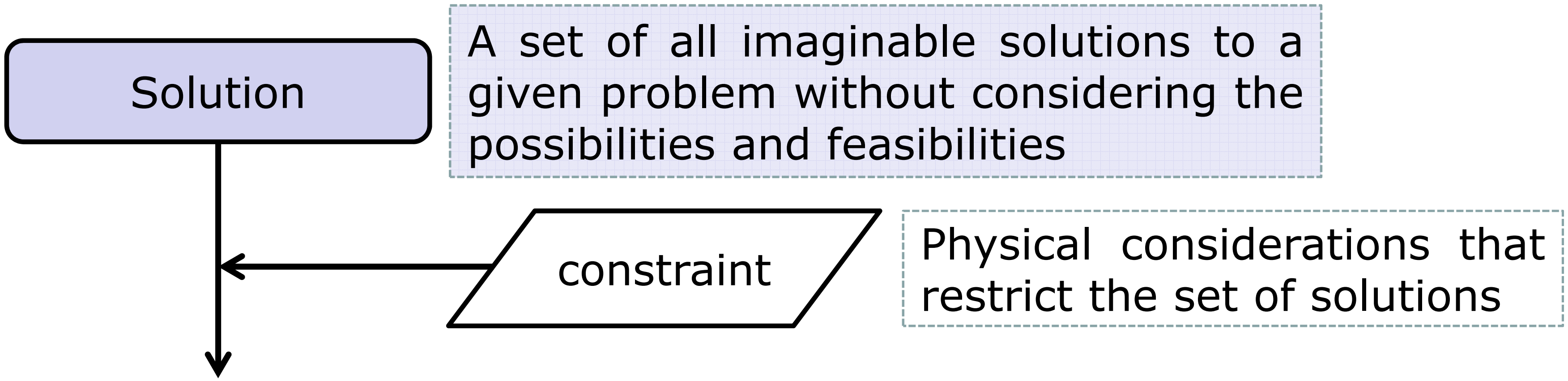
3.0 Basics

- Feasible solution
  - Any subset that satisfies the constraints
- Objective function
  - We are required to find a feasible solution that either maximizes or minimizes a given objective function.
- Optimal solution
  - A feasible solution that maximizes or minimizes an objective function.

3.0 Basics

Solution	A set of all imaginable solutions to a given problem without considering the possibilities and feasibilities

3.0 Basics



Solution
A set of all imaginable solutions to a given problem without considering the possibilities and feasibilities
constraint
Physical considerations that restrict the set of solutions

3.0 Basics

Solution

A set of all imaginable solutions to a given problem without considering the possibilities and feasibilities

constraint

Physical considerations that restrict the set of solutions

Feasible Solution

Any subset that satisfies the constraints

3.0 Basics

Solution

A set of all imaginable solutions to a given problem without considering the possibilities and feasibilities

constraint

Physical considerations that restrict the set of solutions

Feasible Solution

Any subset that satisfies the constraints

Objective function

- A set of functions that measure the satisfaction of the solution
- The measurement can be either profit (to maximize) or cost (to minimize)

3.0 Basics

Solution

A set of all imaginable solutions to a given problem without considering the possibilities and feasibilities

constraint

Physical considerations that restrict the set of solutions

Feasible Solution

Any subset that satisfies the constraints

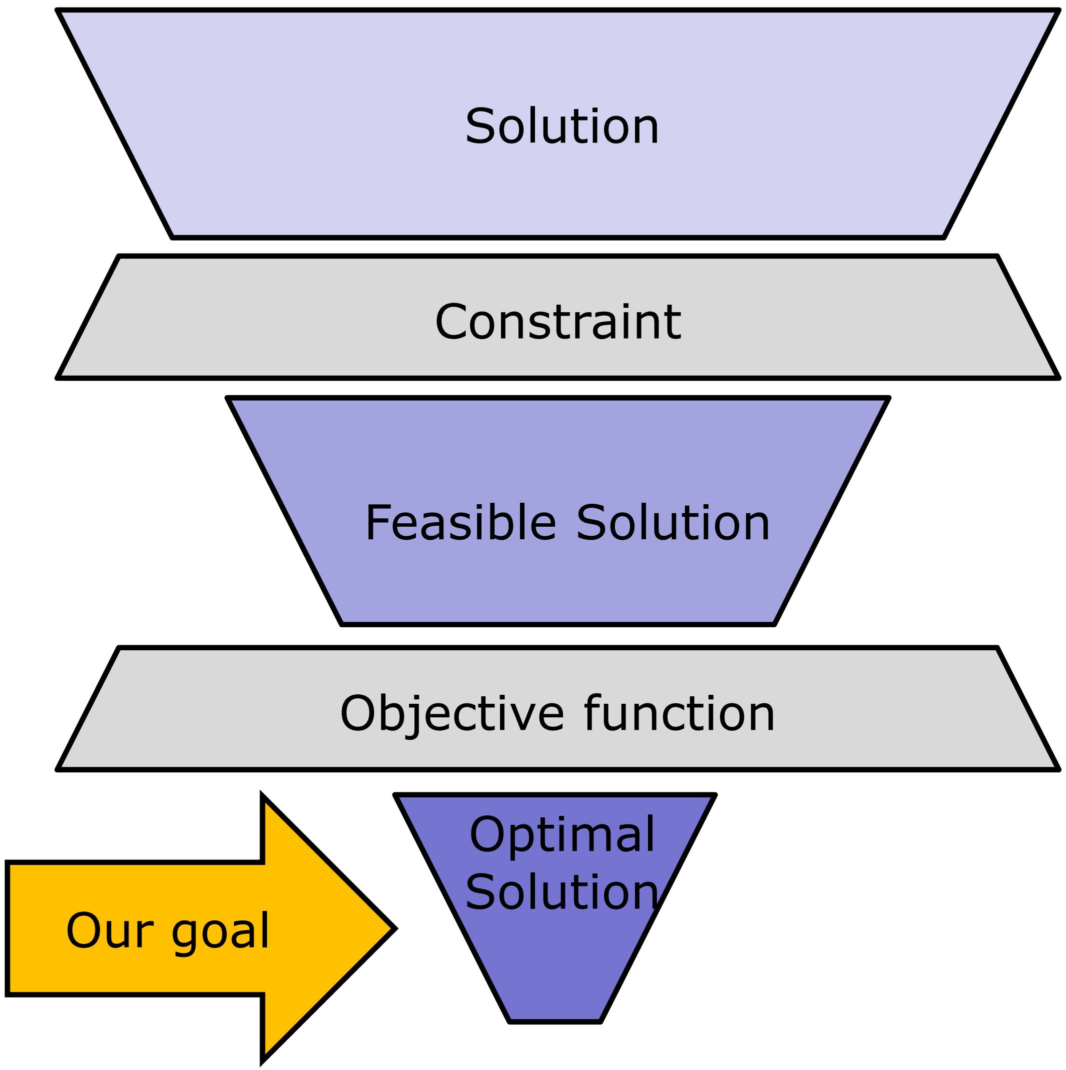
Objective function

- A set of functions that measure the satisfaction of the solution
- The measurement can be either profit (to maximize) or cost (to minimize)

Optimal Solution

A feasible solution that optimizes the objective function

3.0 Basics

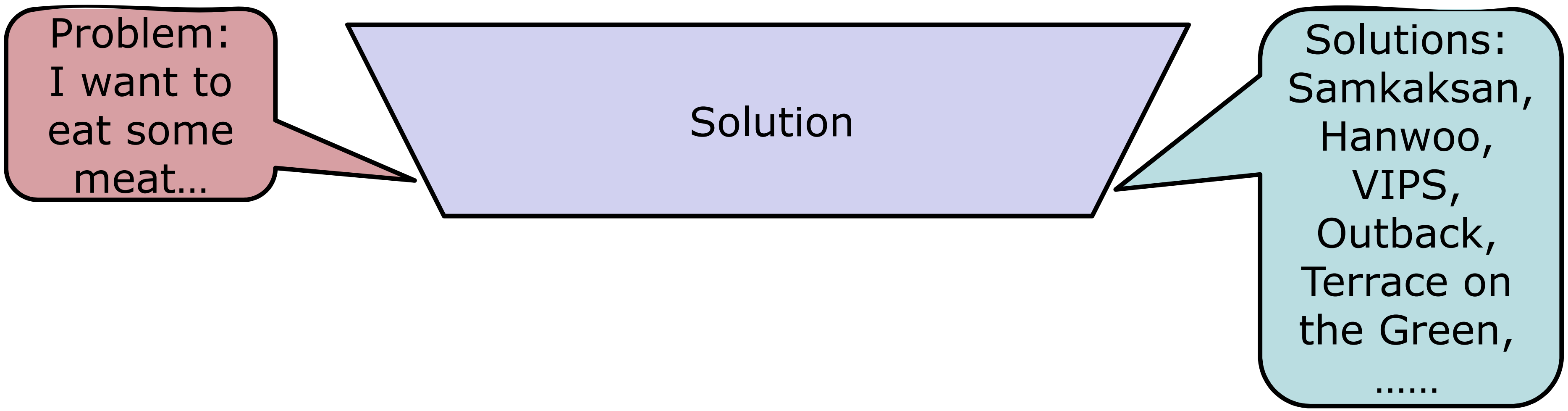


Solution
Constraint
Feasible Solution
Objective function
Our goal
Optimal Solution

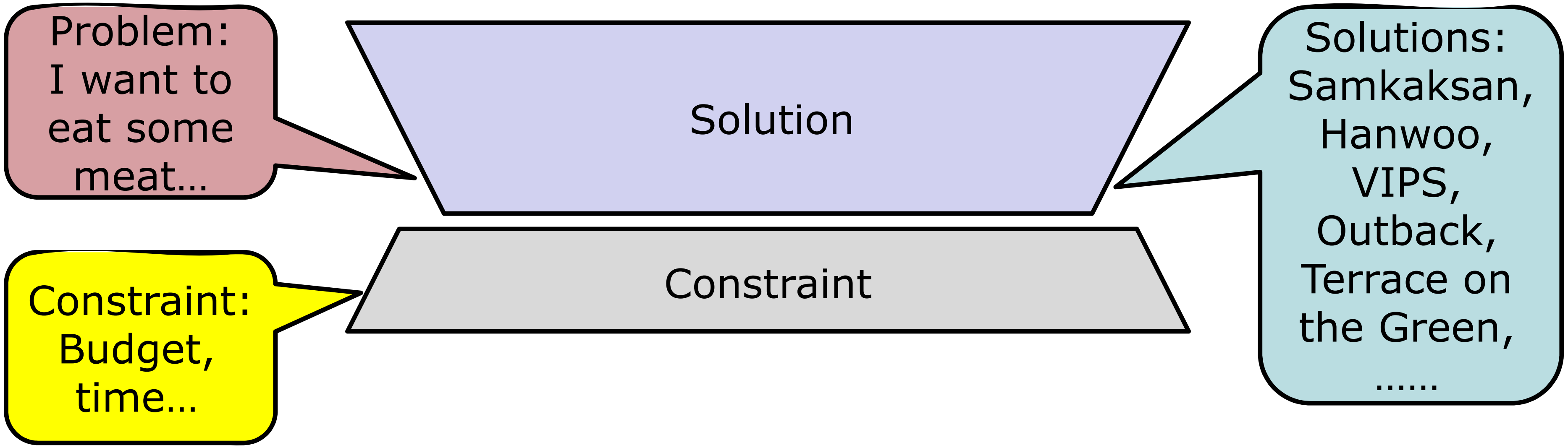
3.0 Basics

Problem:
I want to
eat some
meat…

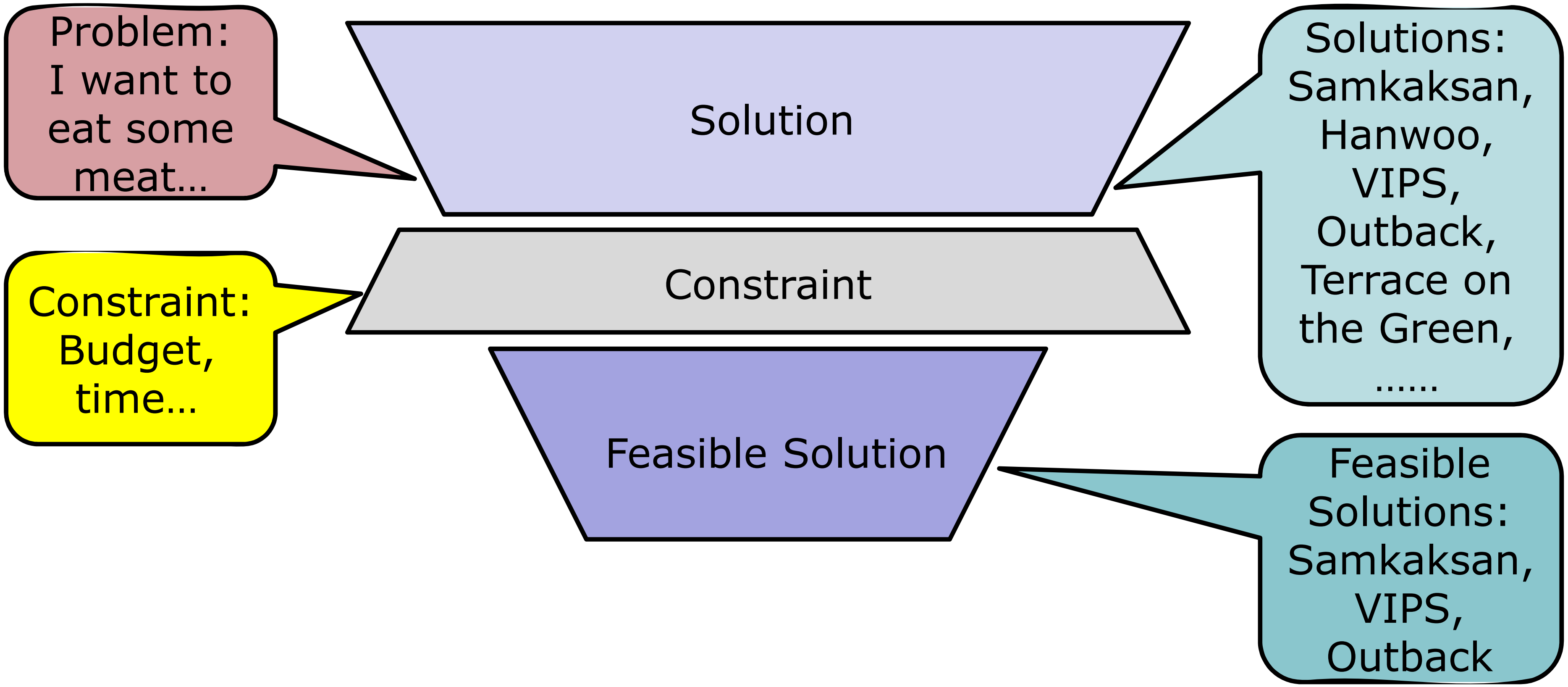
3.0 Basics


Problem:
I want to eat some meat…
Solution
Solutions:
Samkaksan,
Hanwoo,
VIPS,
Outback,
Terrace on the Green,
……

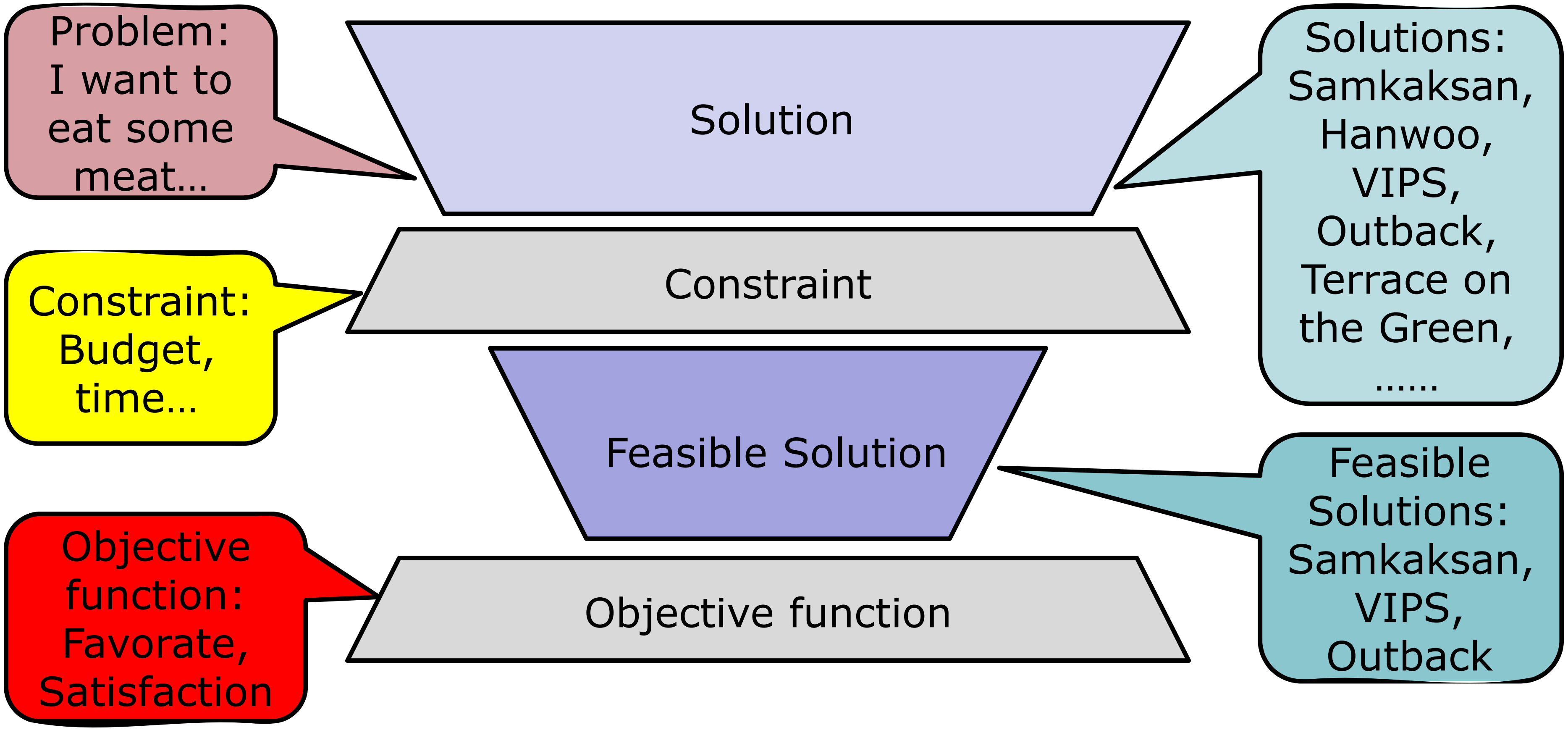
3.0 Basics


Problem: I want to eat some meat…
Solution
Solutions: Samkaksan, Hanwoo, VIPS, Outback, Terrace on the Green, ……
Constraint: Budget, time…
Constraint

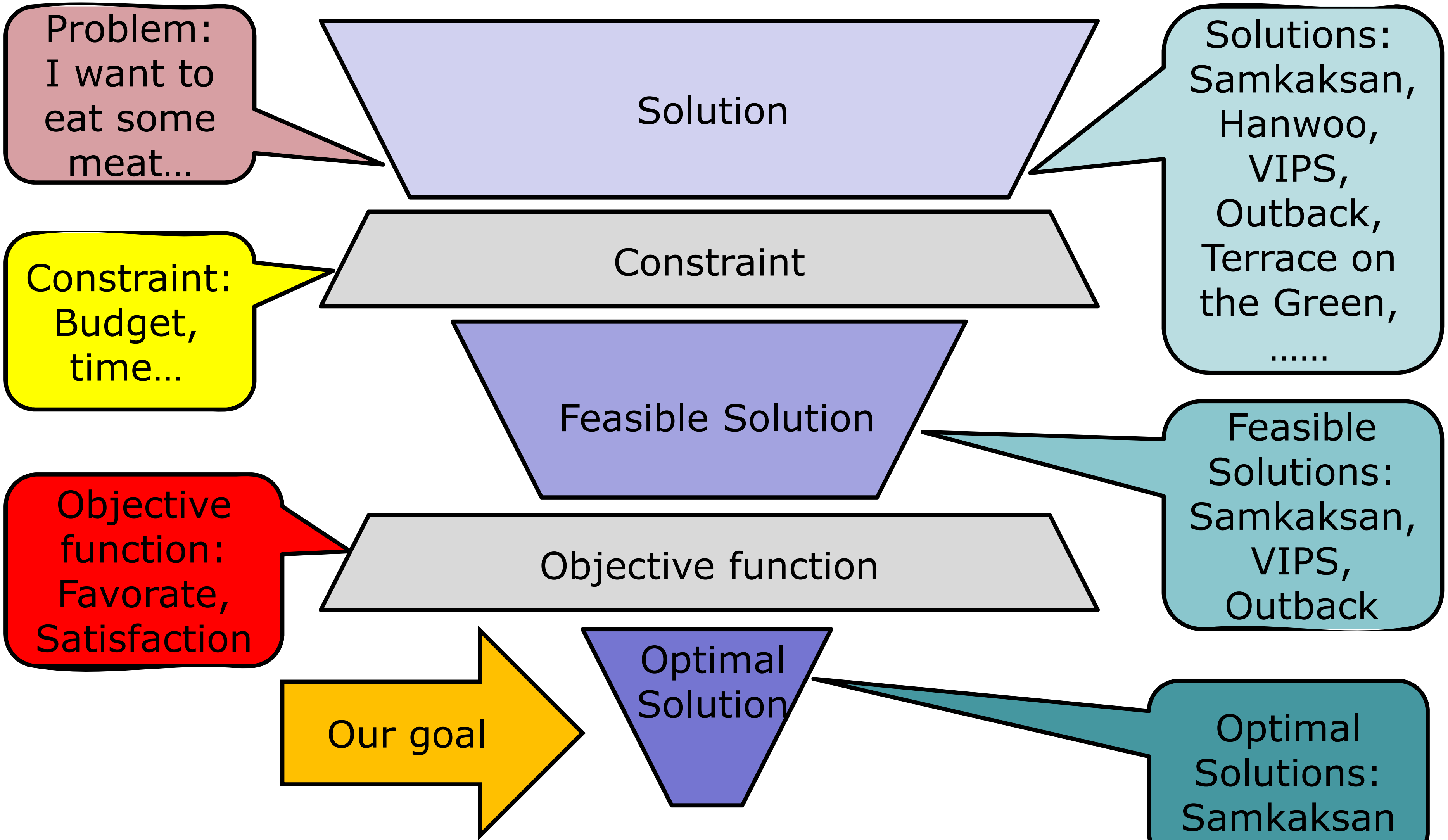
3.0 Basics


Problem: I want to eat some meat…
Solution
Solutions: Samkaksan, Hanwoo, VIPS, Outback, Terrace on the Green, ……
Constraint: Budget, time…
Constraint
Feasible Solution
Feasible Solutions: Samkaksan, VIPS, Outback

3.0 Basics


Problem: I want to eat some meat…
Solution
Solutions: Samkaksan, Hanwoo, VIPS, Outback, Terrace on the Green, ……
Constraint: Budget, time…
Constraint
Feasible Solution
Feasible Solutions: Samkaksan, VIPS, Outback
Objective function: Favorate, Satisfaction
Objective function

3.0 Basics



Problem: I want to eat some meat…
Solution
Solutions: Samkaksan, Hanwoo, VIPS, Outback, Terrace on the Green, ……
Constraint: Budget, time…
Constraint
Feasible Solution
Feasible Solutions: Samkaksan, VIPS, Outback
Objective function: Favorate, Satisfaction
Objective function
Our goal
Optimal Solution
Optimal Solutions: Samkaksan

3.0 Basics

- Greedy algorithm
  - An algorithm that works in stages, considering one input at a time.
  - Selection
    - At each stage, a decision is made regarding whether or not a particular input is in an optimal solution.
    - Selection criteria is based on optimization measure.

3.0 Basics

- Greedy algorithm

```
Greedy( int n, int A[] )
{
    solution ← Φ;
    for ( i = 1 to n )
        x ← SELECT (A);
        if ( FEASIBLE ( solution, x ) )
            solution ← UNION (solution, x);

    return solution;
}
```

3.1 Minimum cost spanning tree

- Spanning tree
  - Connecting every vertex of a graph G using minimum number of edges
  - If $|V| = n$, then at least (n-1) edges are required.
  - Depth-first spanning tree
  - Breadth-first spanning tree

3.1 Minimum cost spanning tree

- Minimum cost spanning tree
  - Computed on a weighted graph
  - A spanning tree whose sum of edge weights is minimum.
  - Properties
    - No new edges.
    - (n-1) edges for |V| = n.
    - No cycle.

3.1 Minimum cost spanning tree

Problem: Connect all the vertices

Solution

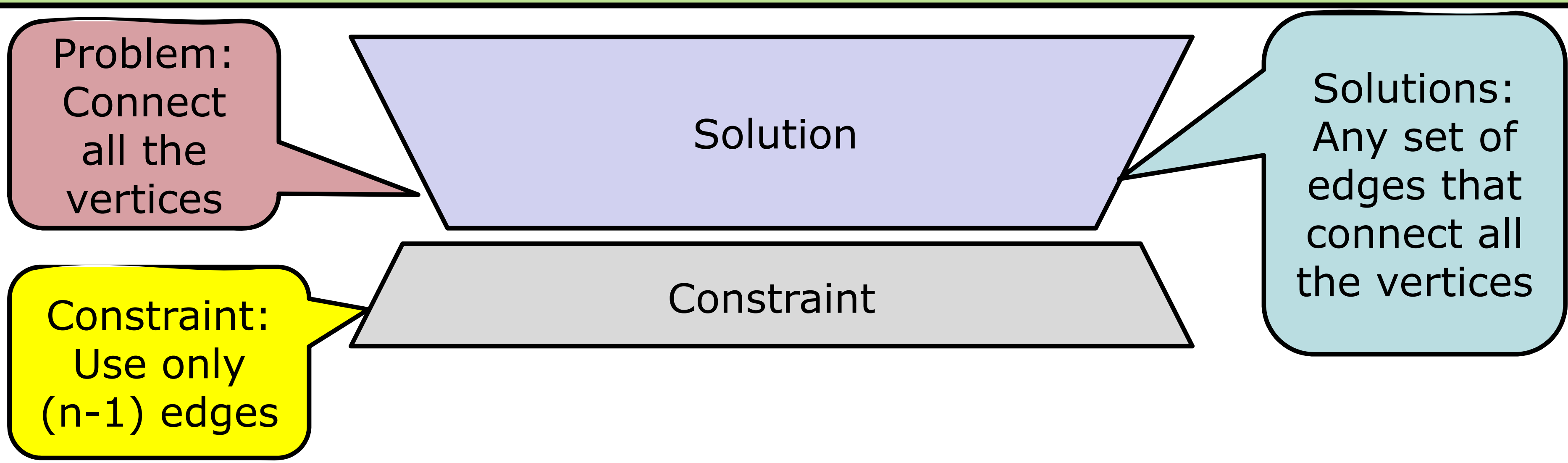
3.1 Minimum cost spanning tree

Problem: Connect all the vertices

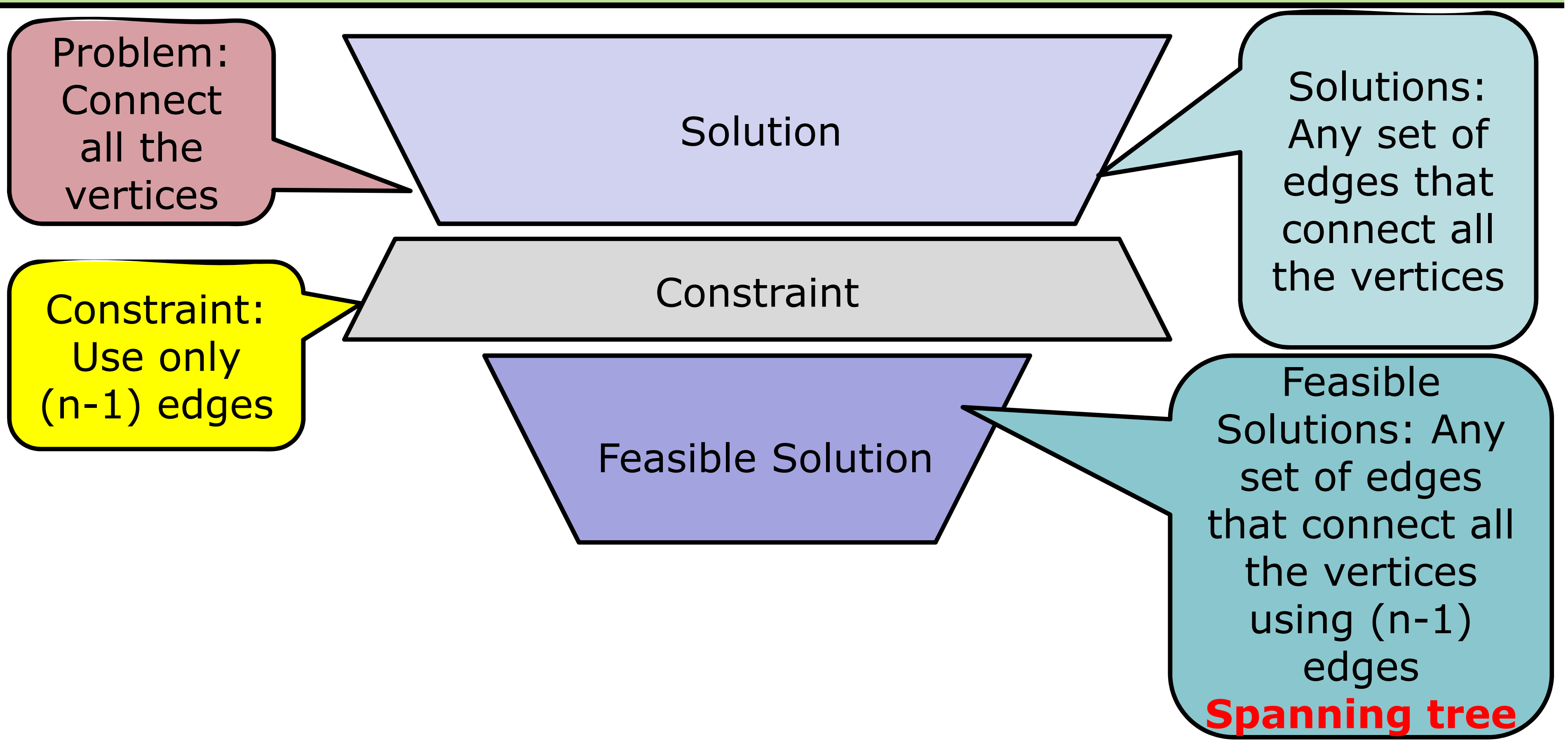
Solution

Solutions: Any set of edges that connect all the vertices

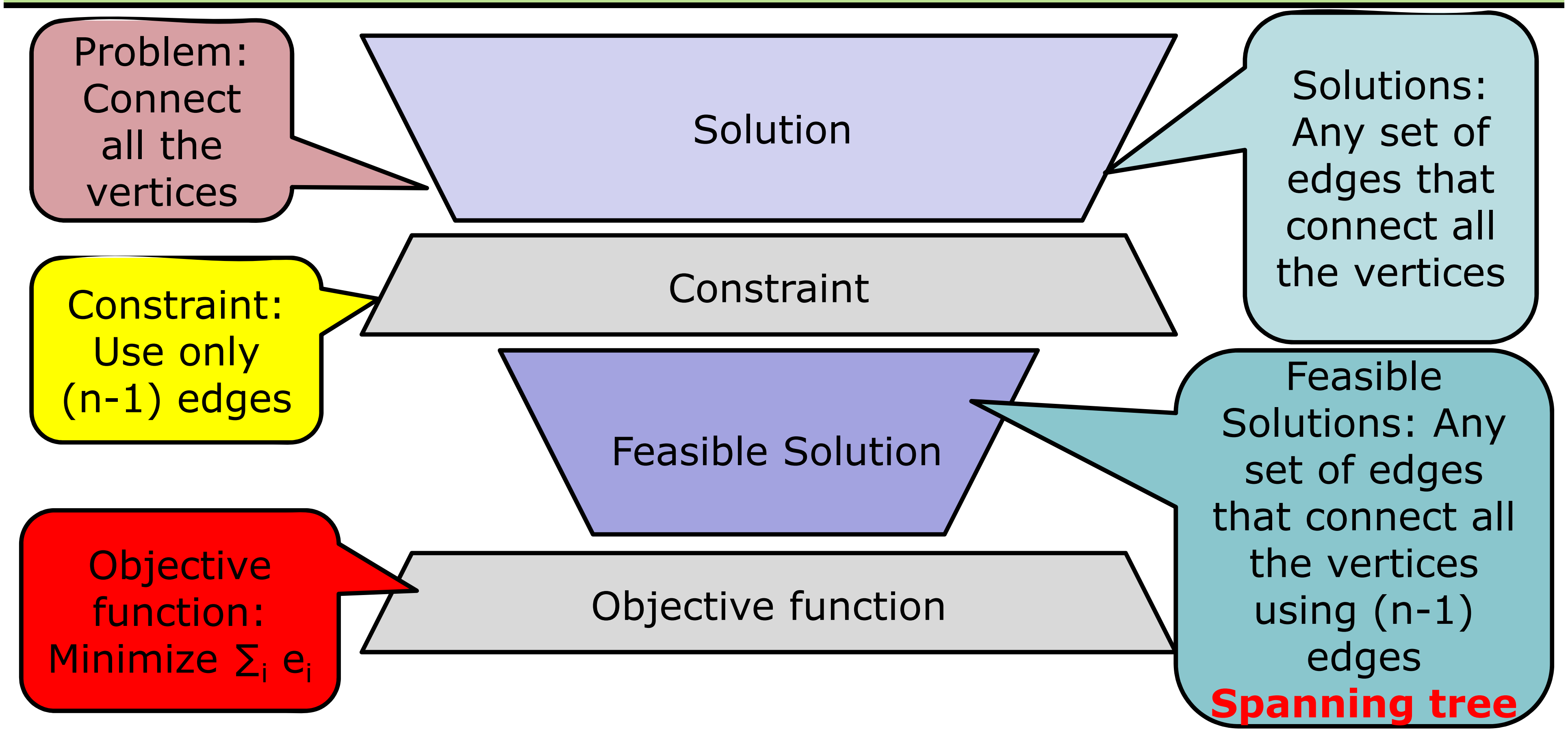
3.1 Minimum cost spanning tree



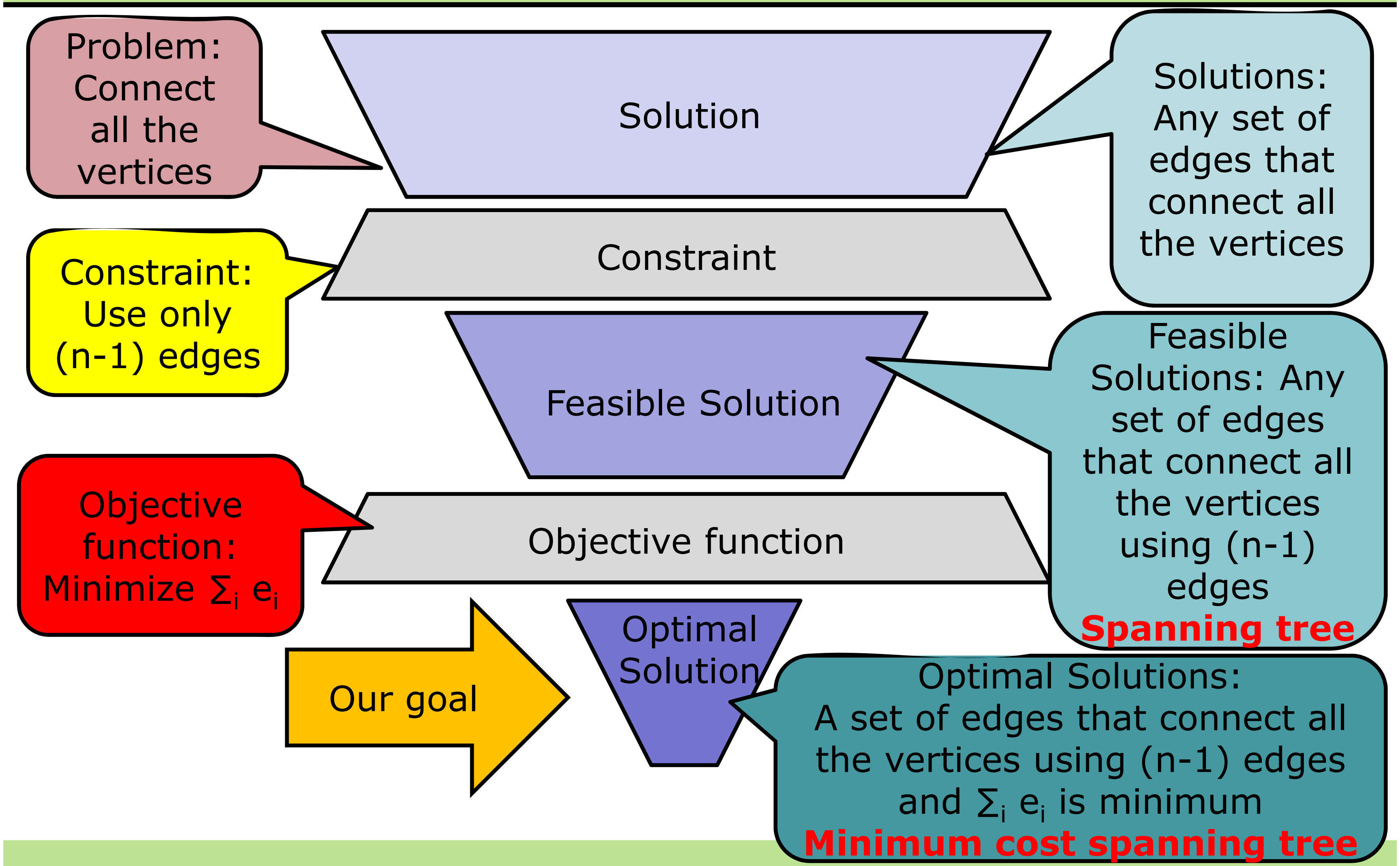
3.1 Minimum cost spanning tree



3.1 Minimum cost spanning tree

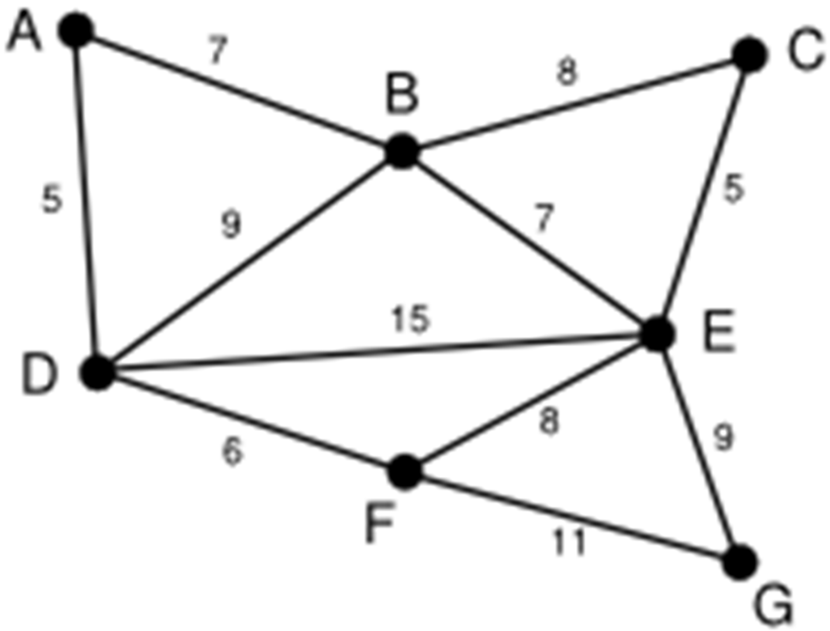


3.1 Minimum cost spanning tree



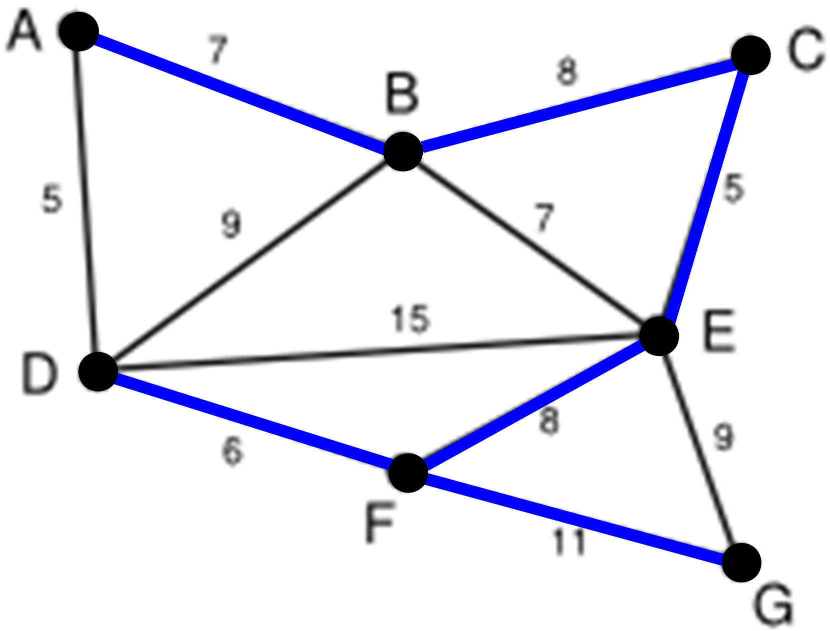
3.1 Minimum cost spanning tree

- Ex)



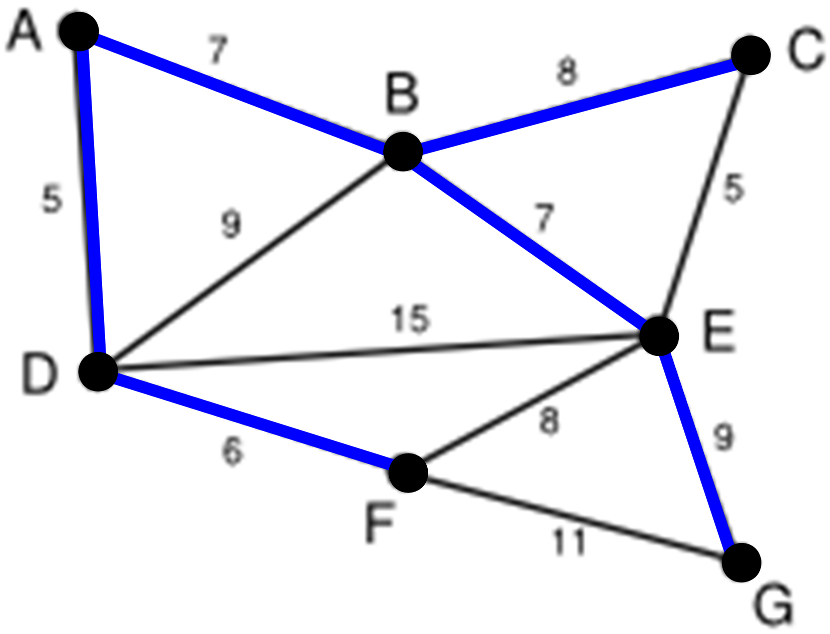
3.1 Minimum cost spanning tree

- Ex) Depth-first spanning tree



3.1 Minimum cost spanning tree

- Ex) Breadth-first spanning tree



3.1.1 Kruskal's Algorithm

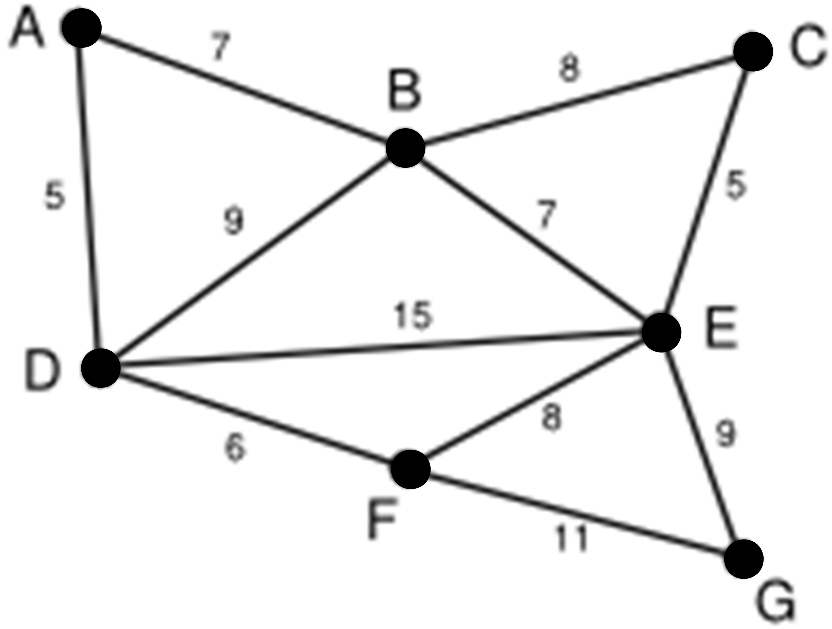
- Edge-oriented algorithm
- Greedy algorithm: 朝四暮三
- Algorithm
  - Sort all the edges of a graph and list them in the ascending order.
  - Choose the edge of the minimum cost from the sorted list, and add it to the minimum cost tree T, if it doesn't make a cycle.
  - Repeat this process until T has (n-1) edges or the sorted list becomes empty.

3.1.1 Kruskal’s Algorithm

```
Tree Kruskal( Vertex V, Edge E )
{
    T = {};
    sort edges in E in ascending order;

    while ( T has less than n-1 edges &&& E is not empty ) {
       choose a least cost edge (v, w) from E;
       delete (v, w) from E;
       if ( (v, w) does not create a cycle in T )
           add (v, w) to T;
       else
           discard (v, w);
    }
    if ( T has fewer than n-1 edges )
       printf(“No Spanning Tree\n” );

    return T;
}
```

3.1.1 Kruskal's Algorithm

- Ex)

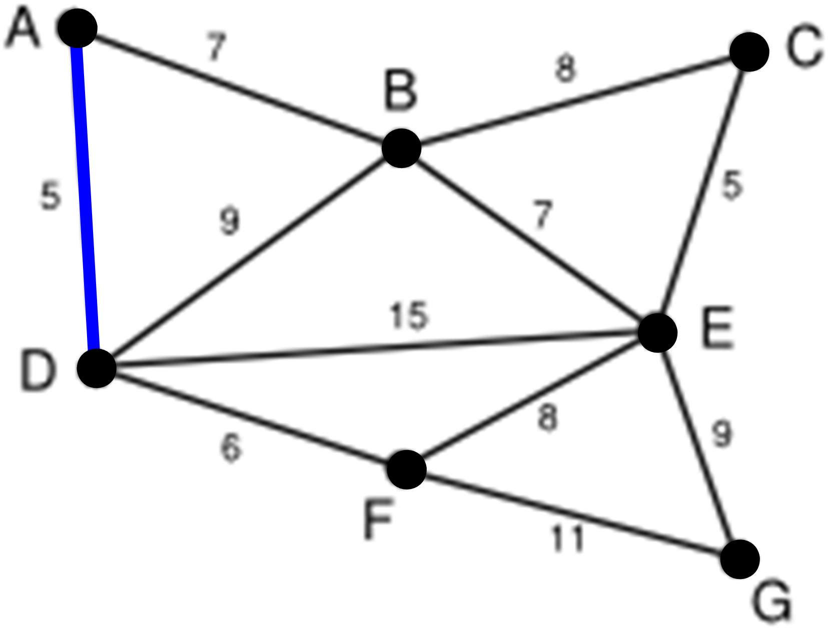


edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T

3.1.1 Kruskal’s Algorithm

- Ex)

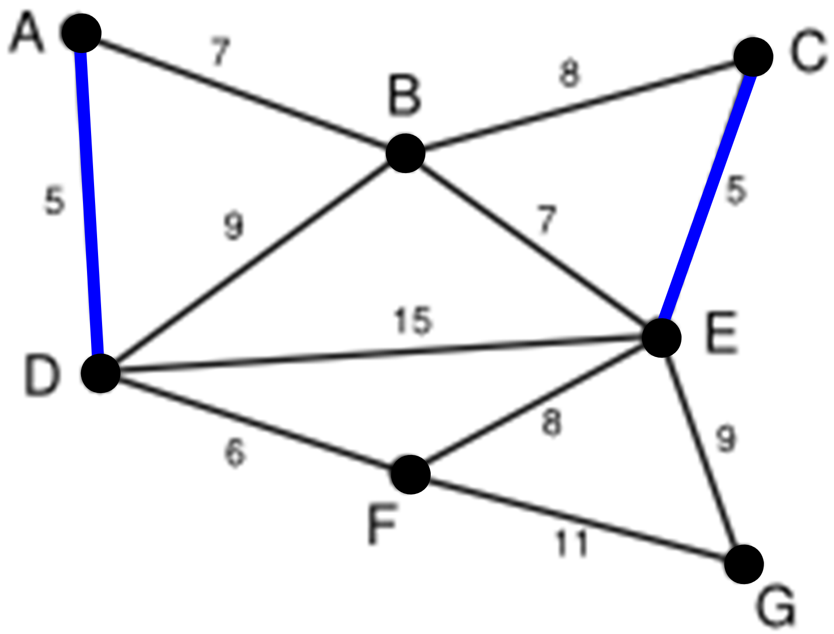


edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)

3.1.1 Kruskal's Algorithm

- Ex)

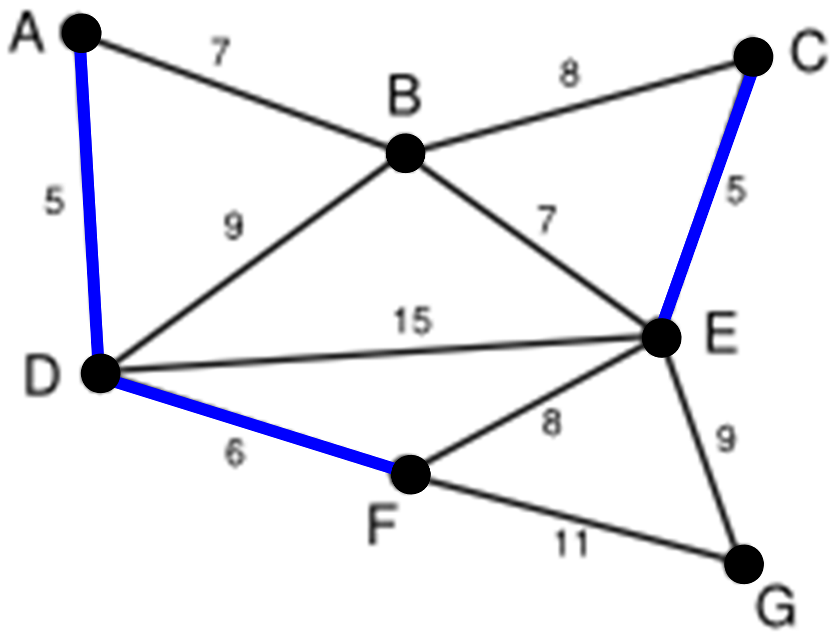


edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)

3.1.1 Kruskal's Algorithm

- Ex)

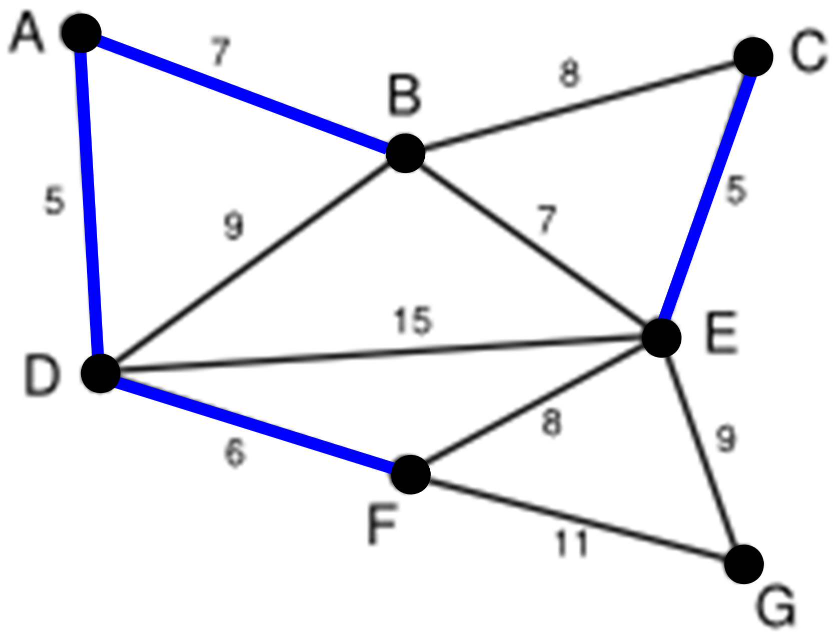


edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)

3.1.1 Kruskal's Algorithm

- Ex)

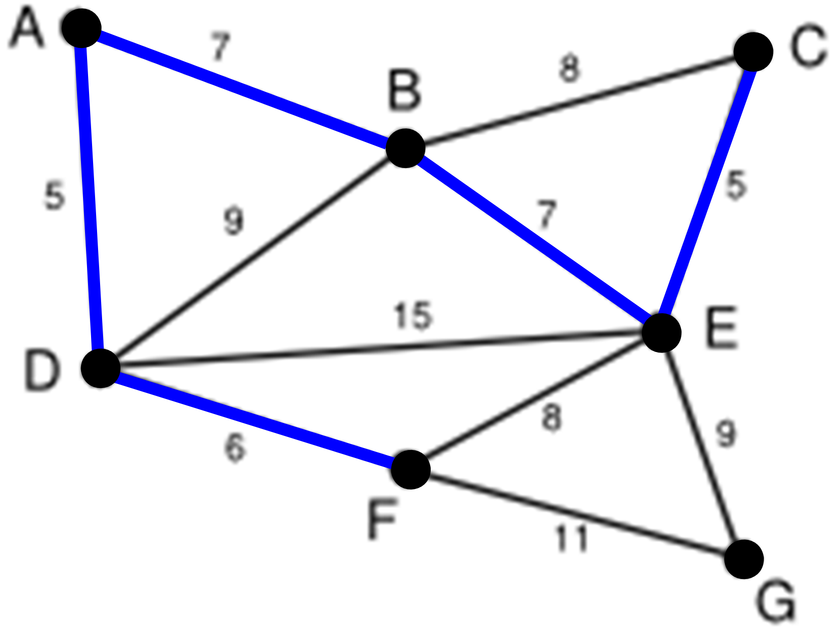


edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)
(A,B)

3.1.1 Kruskal's Algorithm

- Ex)

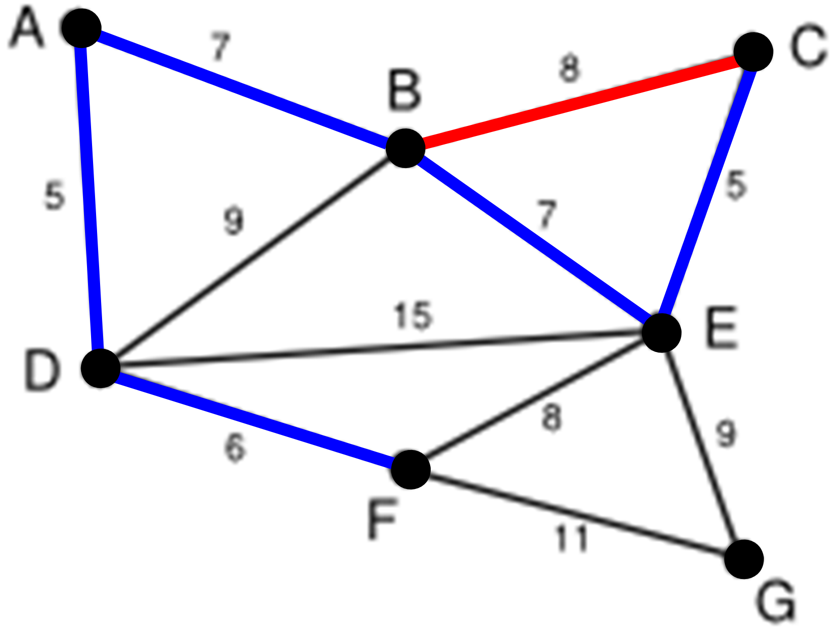


edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)
(A,B)
(B,E)

3.1.1 Kruskal's Algorithm

- Ex)

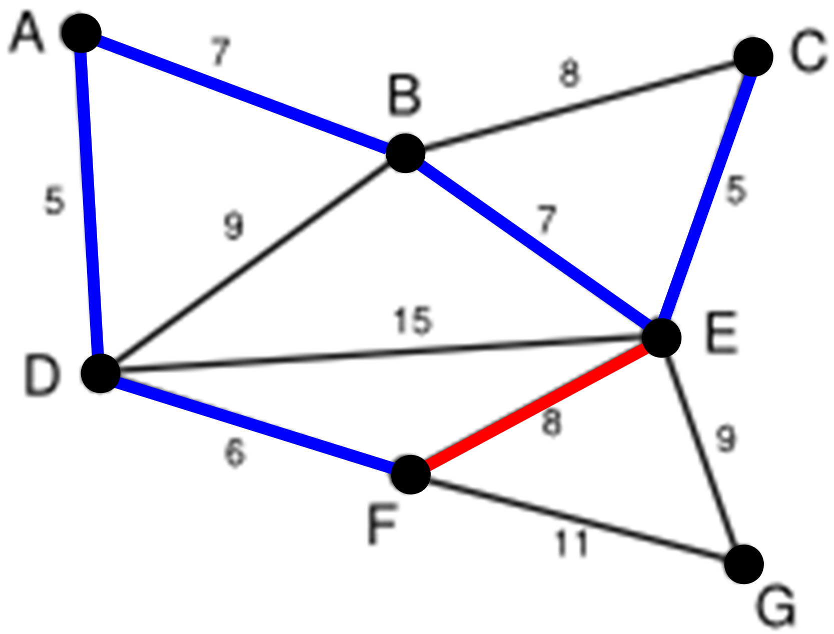


edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)
(A,B)
(B,E)

3.1.1 Kruskal's Algorithm

- Ex)

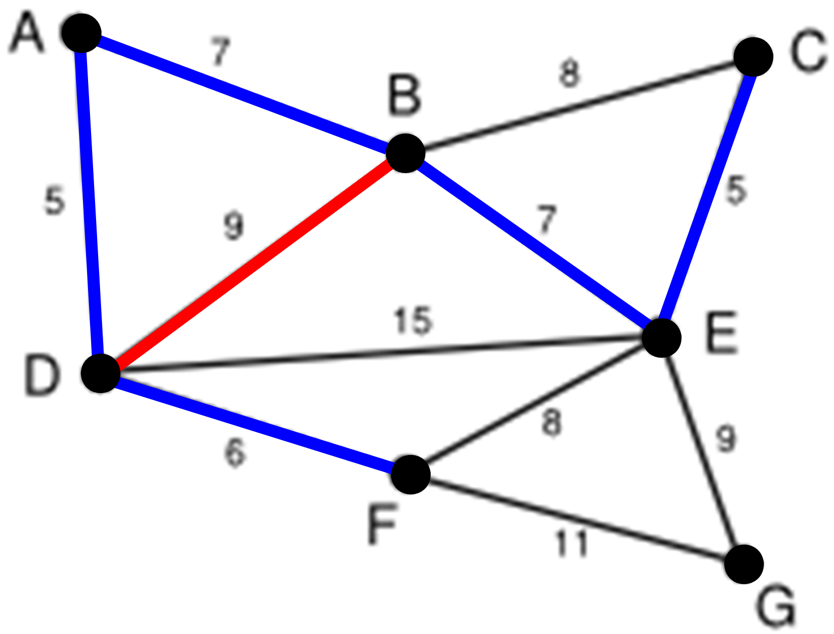


edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)
(A,B)
(B,E)

3.1.1 Kruskal's Algorithm

- Ex)

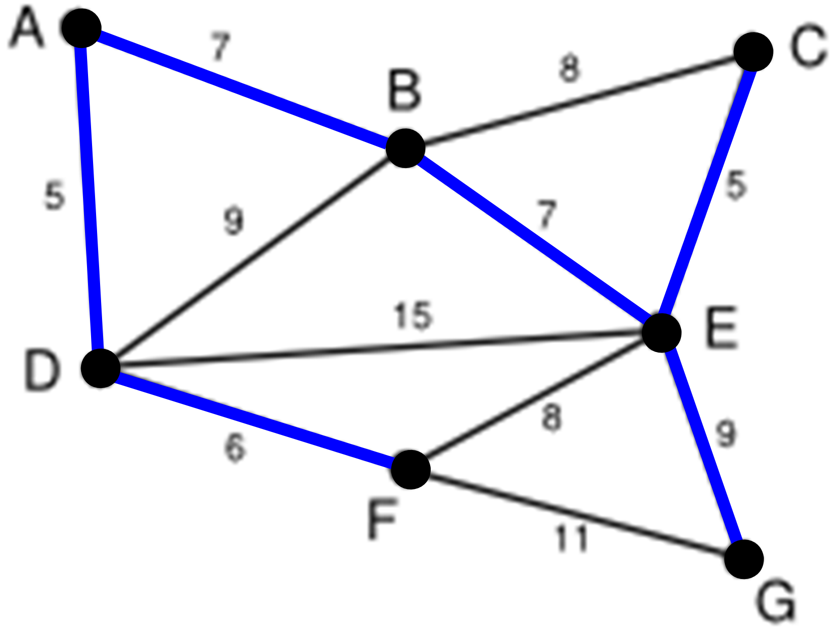


edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)
(A,B)
(B,E)

3.1.1 Kruskal's Algorithm

- Ex)



edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)
(A,B)
(B,E)
(E,G)

3.1.1 Kruskal's Algorithm

- Performance analysis
  - Sorting edges → O(|E| log |E|)
  - Adding edges → O(|V|)
    - Check cycles → O(|V|)

- How to improve performance?
  - Use UNION-FIND operations for checking cycles
  - Assign labels on the vertices for UNION-FIND

3.1.1 Kruskal's Algorithm

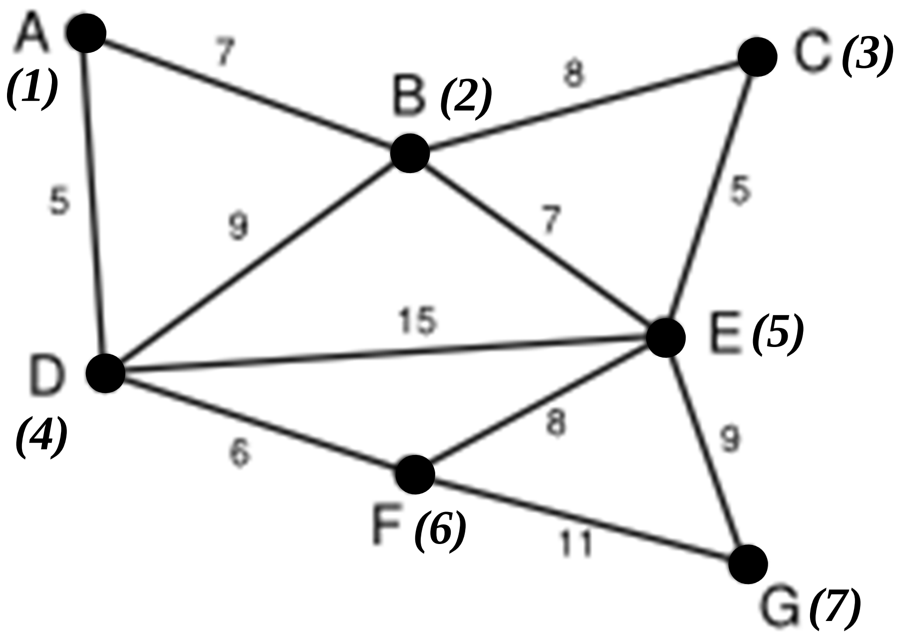
- Another version of Kruskal's algorithm
  - Checking cycle by labeling vertices
    - If two vertices have same labels, then adding the edge that connects the two vertices becomes a cycle

3.1.1 Kruskal's Algorithm

```
Tree Kruskal( Vertex V, Edge E )
{
    T = {};
    sort edges in E in ascending order;

    for each vertex v in the set V
            NEW_LABEL(v);

    for each (u,v) in E, in ascending order of weight {
            if LABEL(u) is not equal to LABEL(v) {
                add the edge (u,v) to the tree T;
          UNION( u, v ); return T;
            }
    }

    return T;
}
```

3.1.1 Kruskal's Algorithm

- Ex)

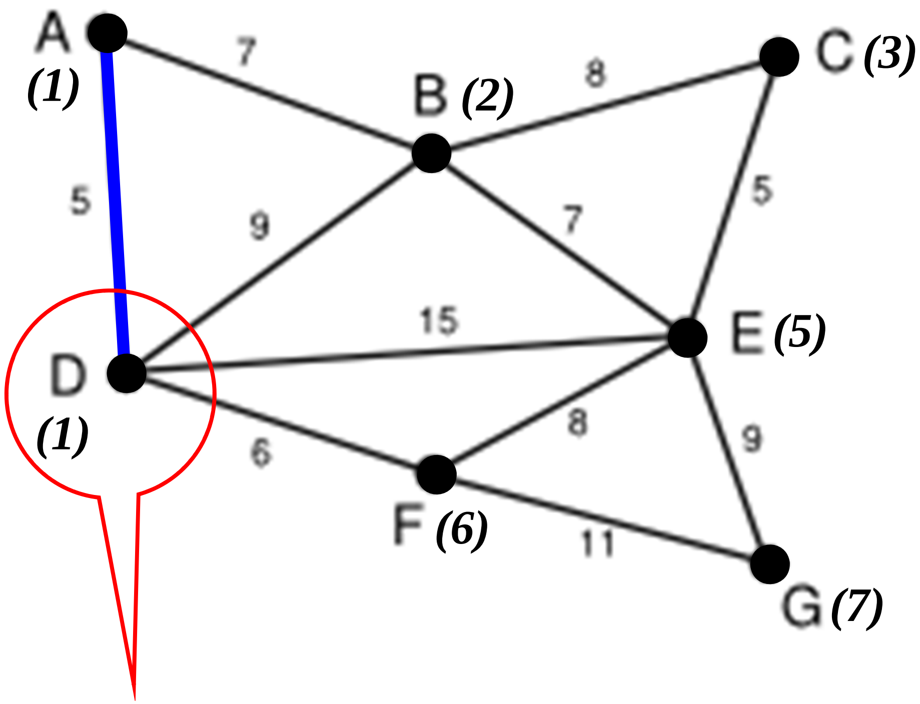


edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T

3.1.1 Kruskal's Algorithm

- Ex)



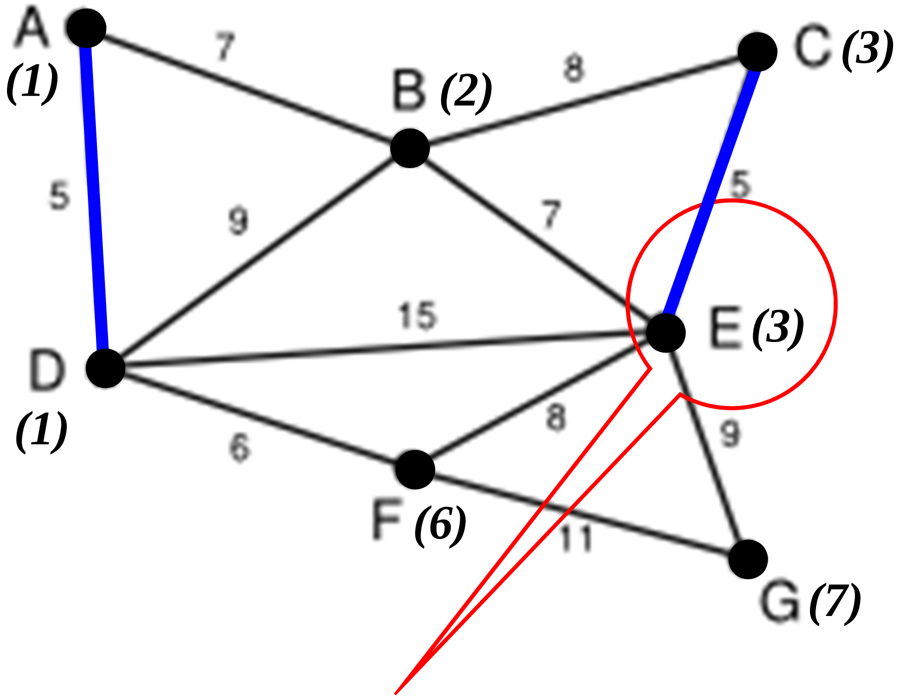
`UNION (A, D)` changes (4) to (1)

edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)

3.1.1 Kruskal's Algorithm

- Ex)



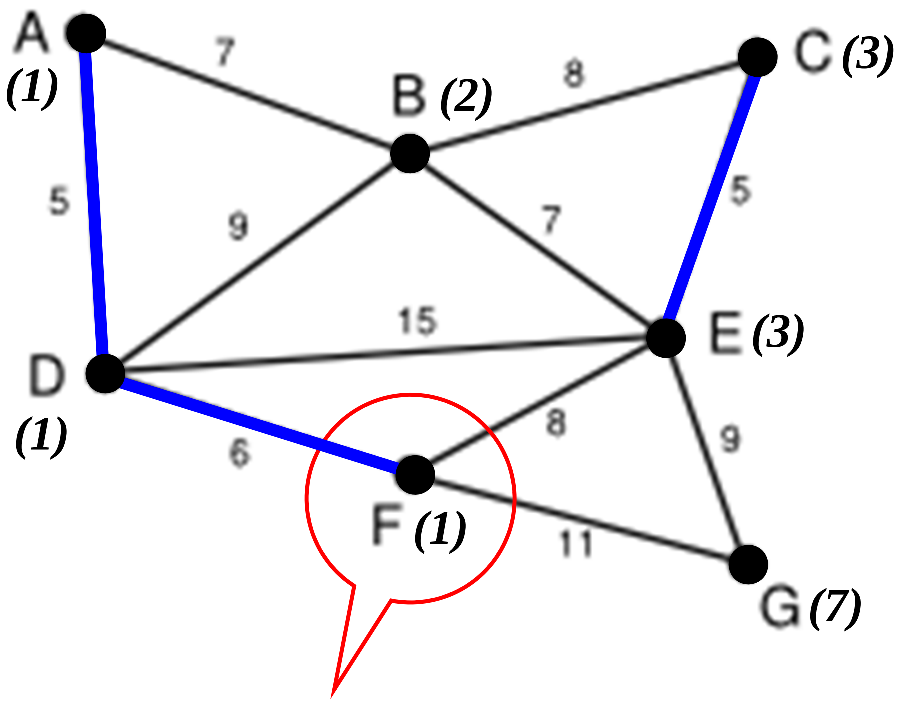
`UNION (C, E)` changes (5) to (3)

edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)

3.1.1 Kruskal's Algorithm

- Ex)



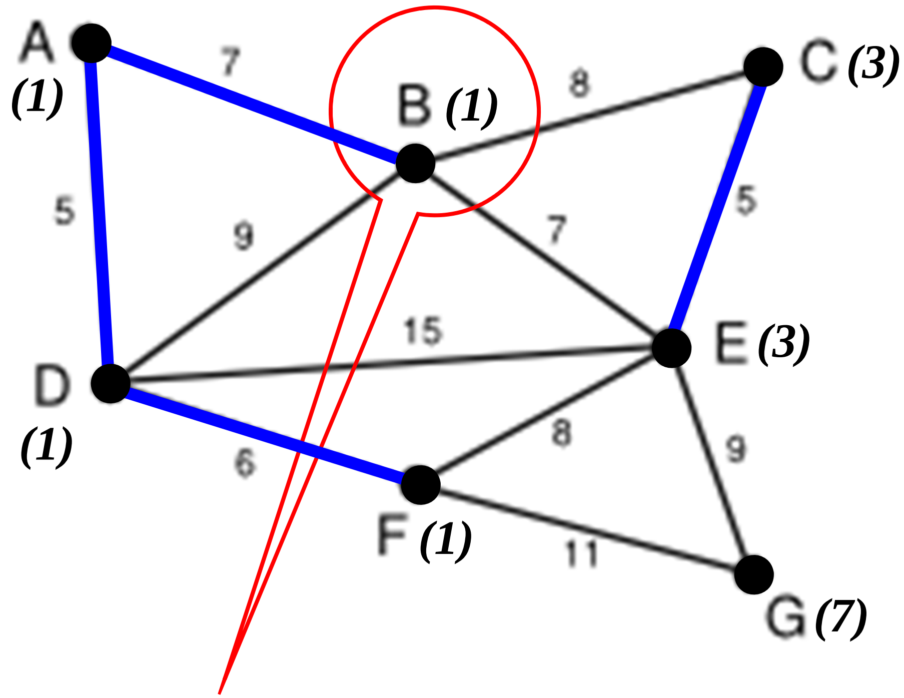
`UNION (D, F)` changes (6) to (1)

edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)

3.1.1 Kruskal's Algorithm

- Ex)



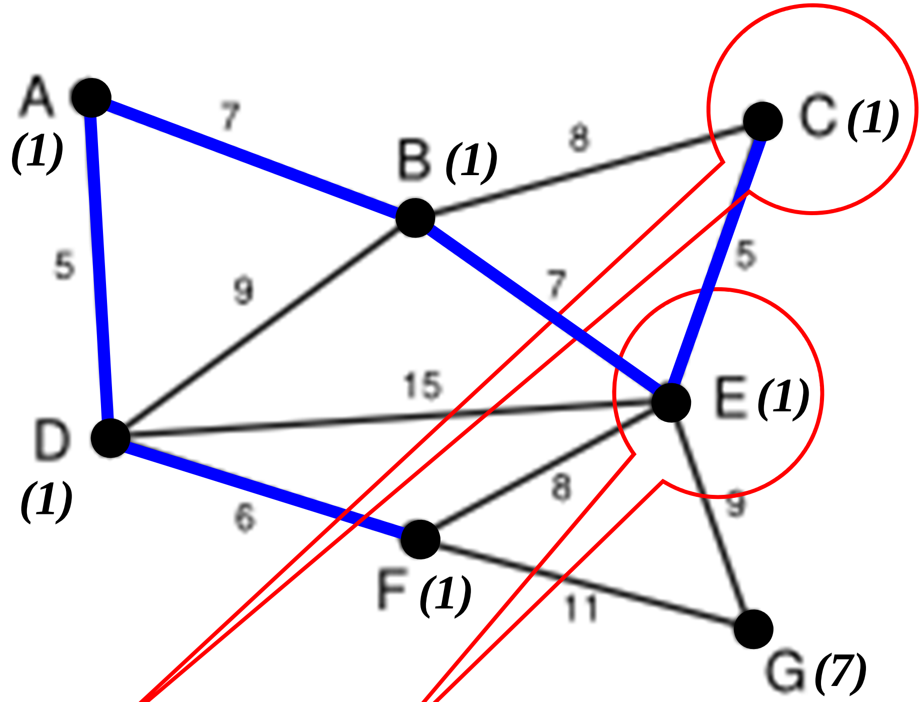
`UNION (A, B)` changes (2) to (1)

edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)
(A,B)

3.1.1 Kruskal's Algorithm

- Ex)



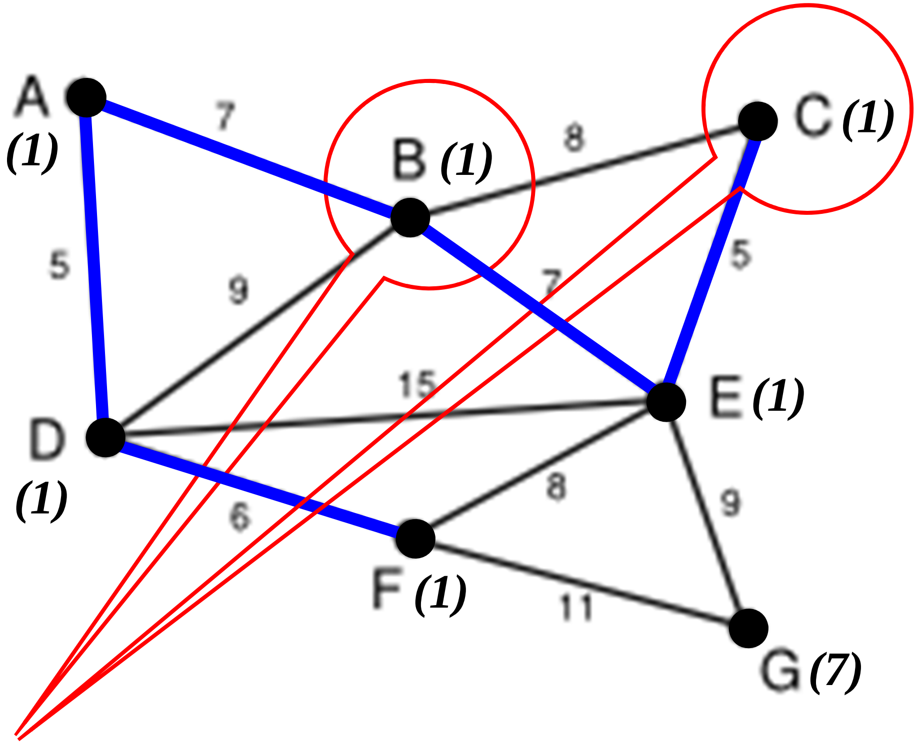
`UNION (B, E)` changes (3) to (1)

edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)
(A,B)
(B,E)

3.1.1 Kruskal's Algorithm

- Ex)



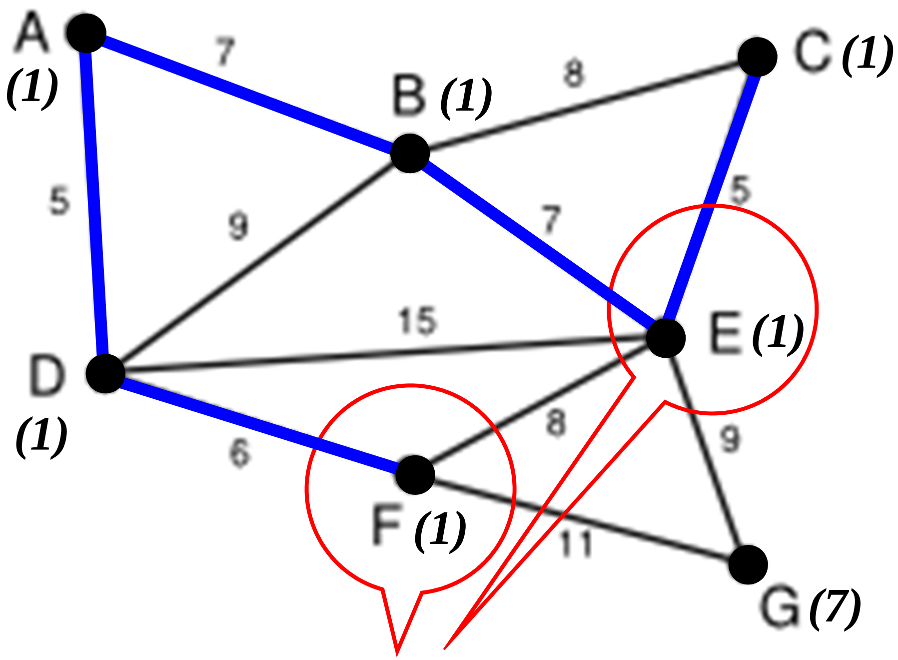
B & C canot be connected,
since both labels are same → cycle

edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)
(A,B)
(B,E)

3.1.1 Kruskal's Algorithm

- Ex)



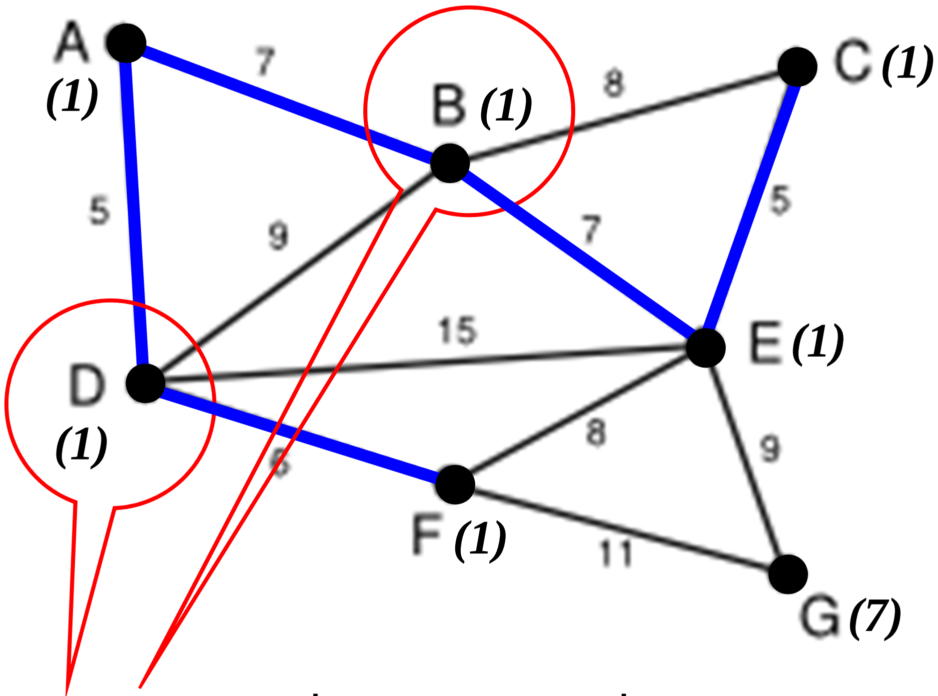
E & F canot be connected,
since both labels are same → cycle

edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)
(A,B)
(B,E)

3.1.1 Kruskal's Algorithm

- Ex)



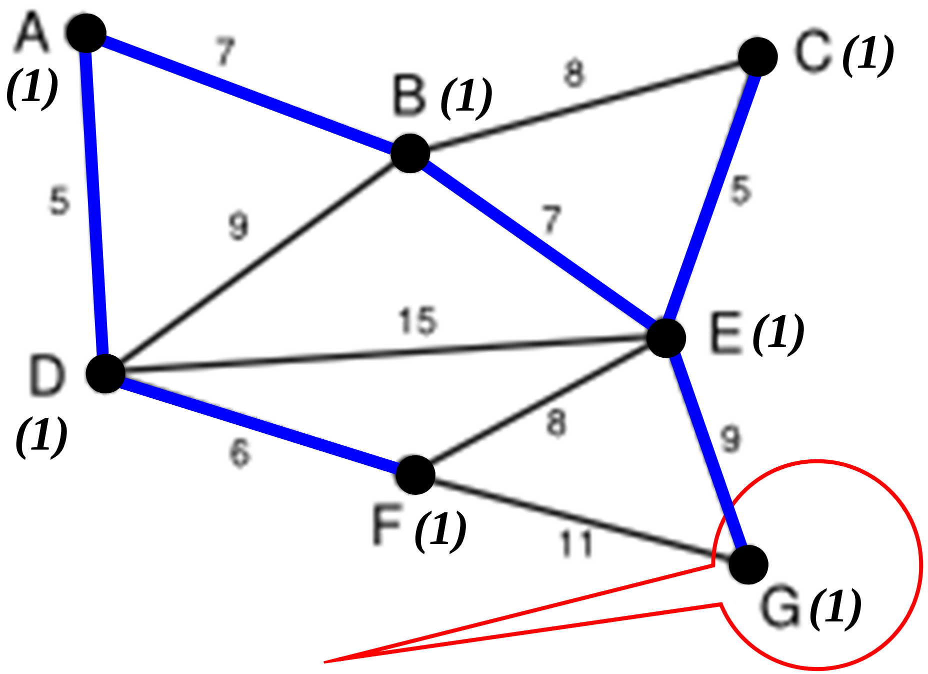
B & D canot be connected,
since both labels are same → cycle

edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)
(A,B)
(B,E)

3.1.1 Kruskal's Algorithm

- Ex)



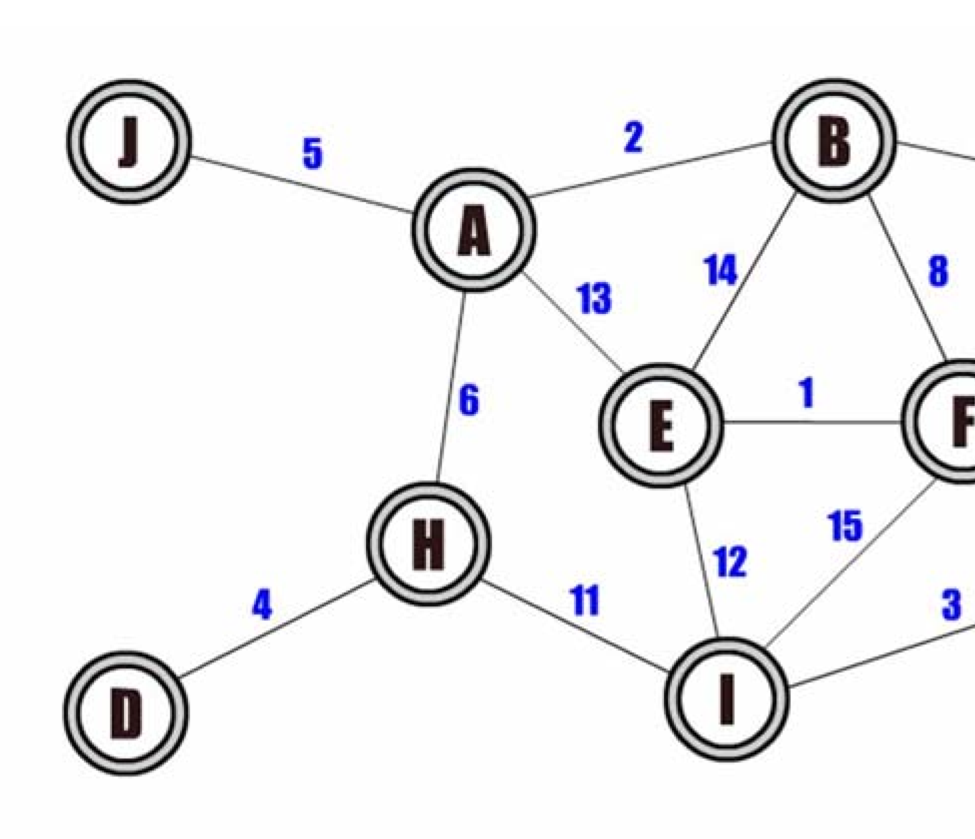
UNION (E, G) changes (7) to (1)

edges	*weights*
(A,D)	5
(C,E)	5
(D,F)	6
(A,B)	7
(B,E)	7
(B,C)	8
(E,F)	8
(B,D)	9
(E,G)	9
(F,G)	11
(D,E)	15

T
(A,D)
(C,E)
(D,F)
(A,B)
(B,E)
(E,G)

3.1.1 Kruskal's Algorithm

- Ex)



3.1.2 Prim's Algorithm

- Vertex-based algorithm
  - Greedy algorithm
  - Vertices on a Graph is classified into three categories:
    - Vertices in minimum-cost spanning tree (T)
    - Vertices incident to the vertices in T
    - Other vertices

3.1.2 Prim's Algorithm

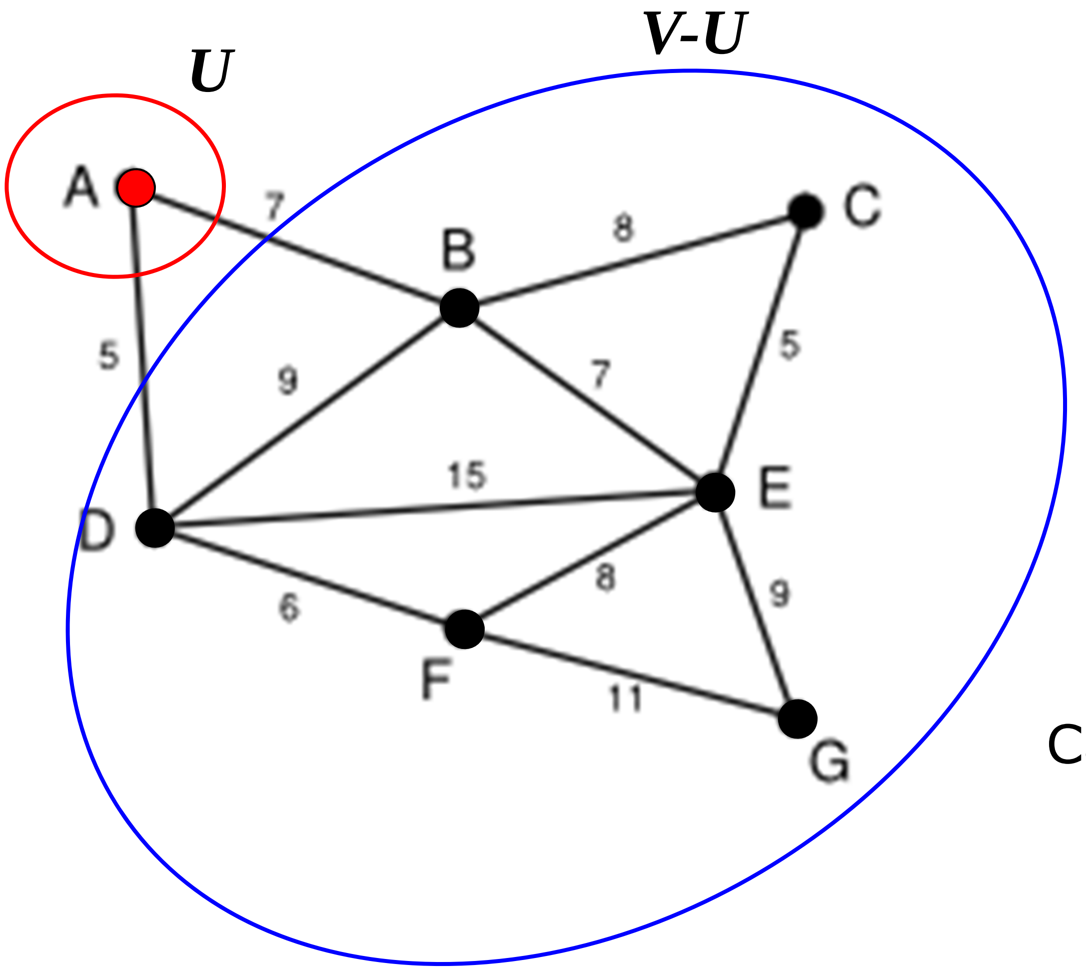
- Algorithm
  - Initially, set T as Φ.
  - Find all the vertices incident to the vertices in T.
  - Find the minimum-weight edge among the edges that connect a vertex belongs to T and a vertex that does not belong to T.
  - If the edge doesn't make a cycle, then add the vertex on the edge to T.
  - Repeat this process until all vertices belong to T.

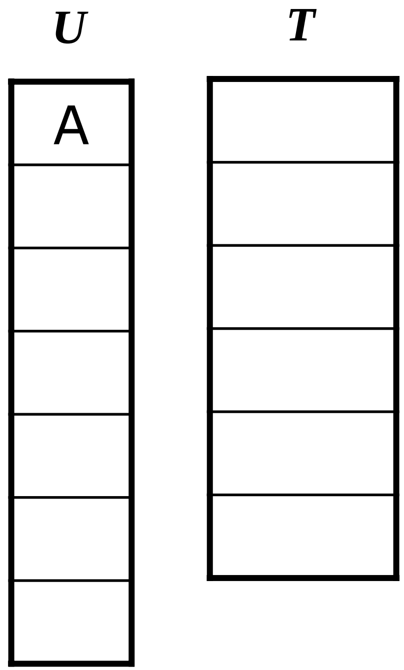
3.1.2 Prim's Algorithm

```
Tree Prim( Vertex V, Edge E )
{
    Vertex *U;
    vertex u,v;
    T = { };
    U = { A };
    while (U != V) {
        (u,v) = lowest cost edge with u in U and v in (V - U);
        T += (u,v);
        U += v;
    }
    return T;
}
```

3.1.2 Prim's Algorithm

- Ex)

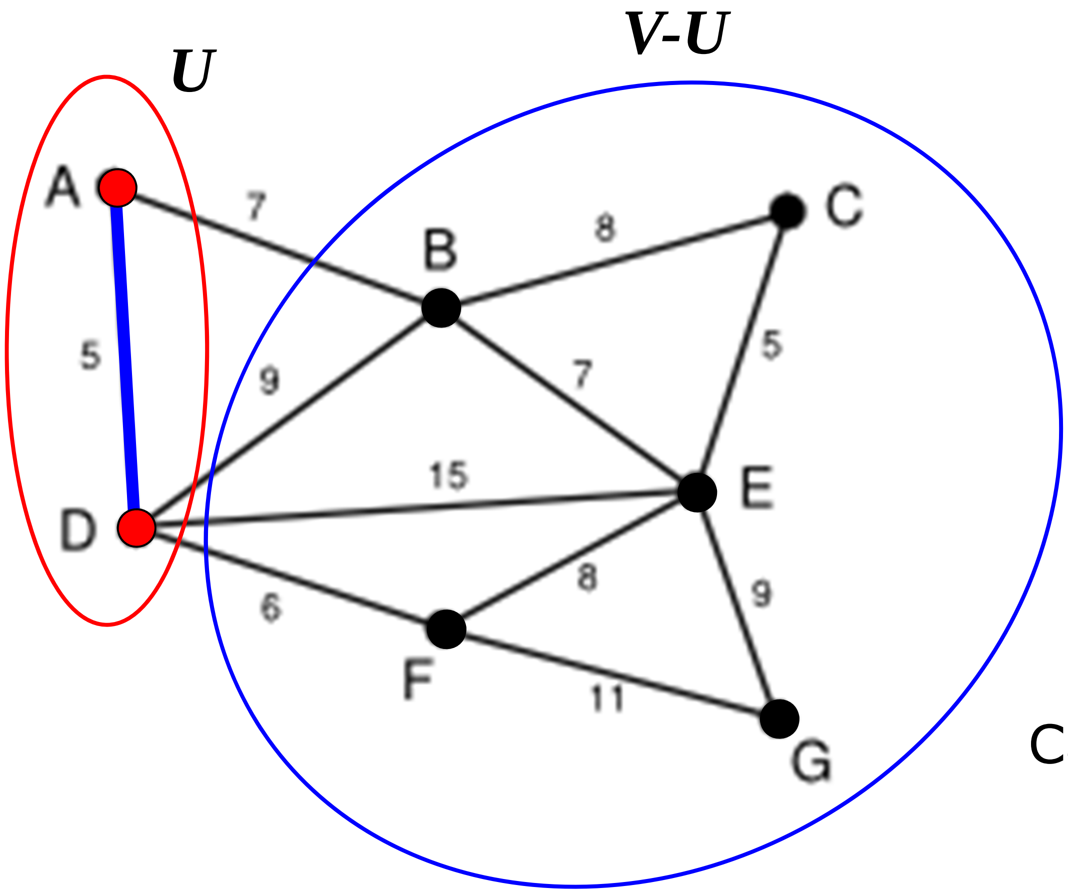




Candidate edges: (A, B): 7
(A, D): 5

3.1.2 Prim's Algorithm

- Ex)



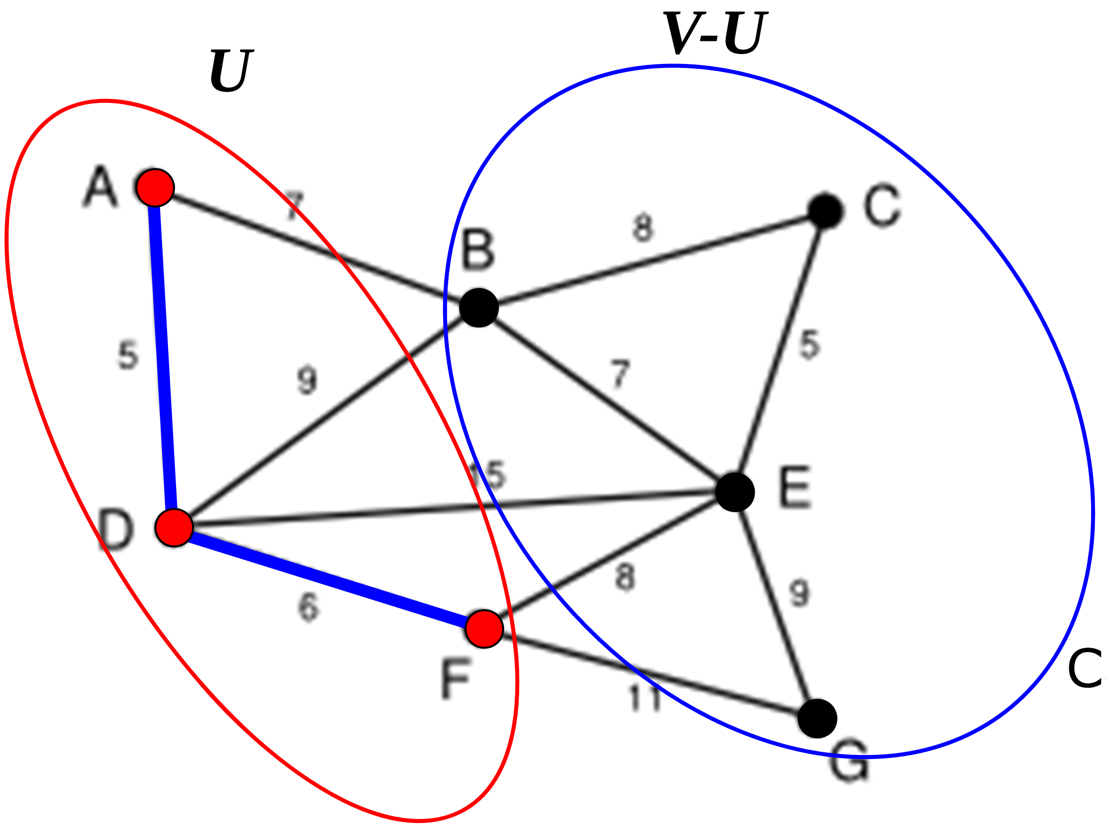
U
A
D

T
(A,D)

Candidate edges: (A, B): 7
(D, B): 9
(D, E): 15
(D, F): 6

3.1.2 Prim's Algorithm

- Ex)



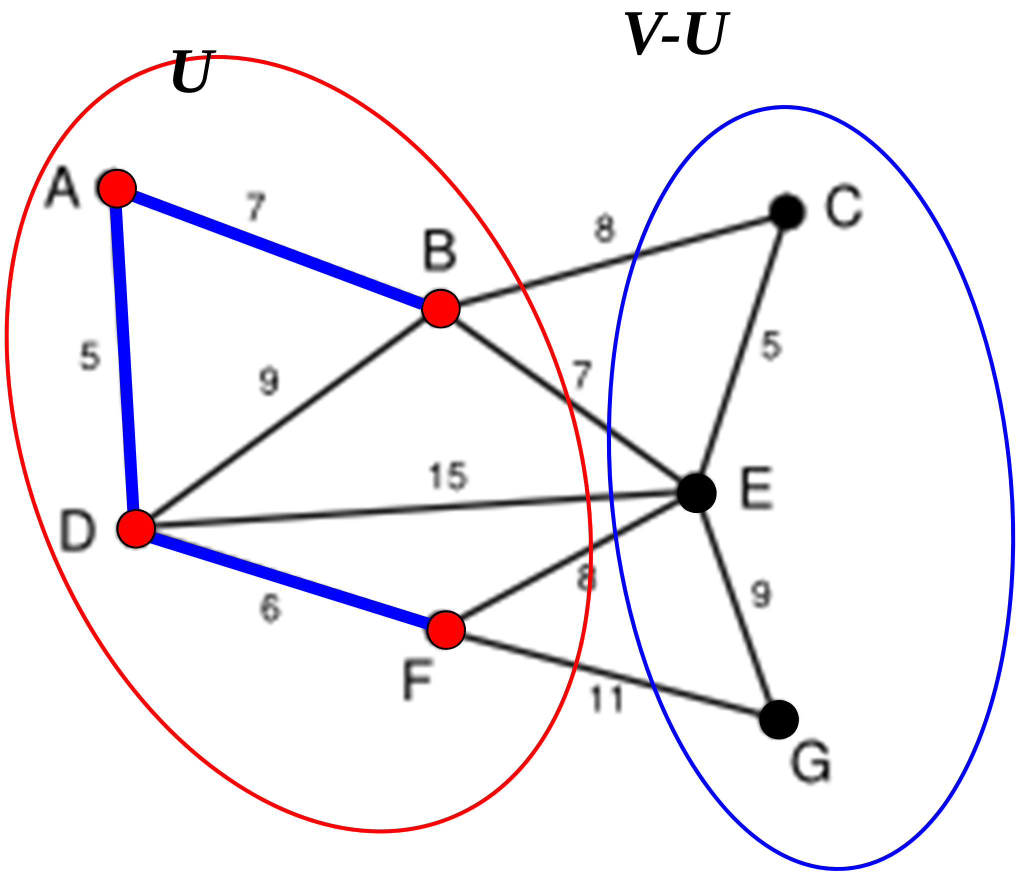
U
A
D
F

T
(A,D)
(D,F)

Candidate edges: (A, B): 7
(D, B): 9
(D, E): 15
(F, E): 8
(F, G): 11

3.1.2 Prim's Algorithm

- Ex)



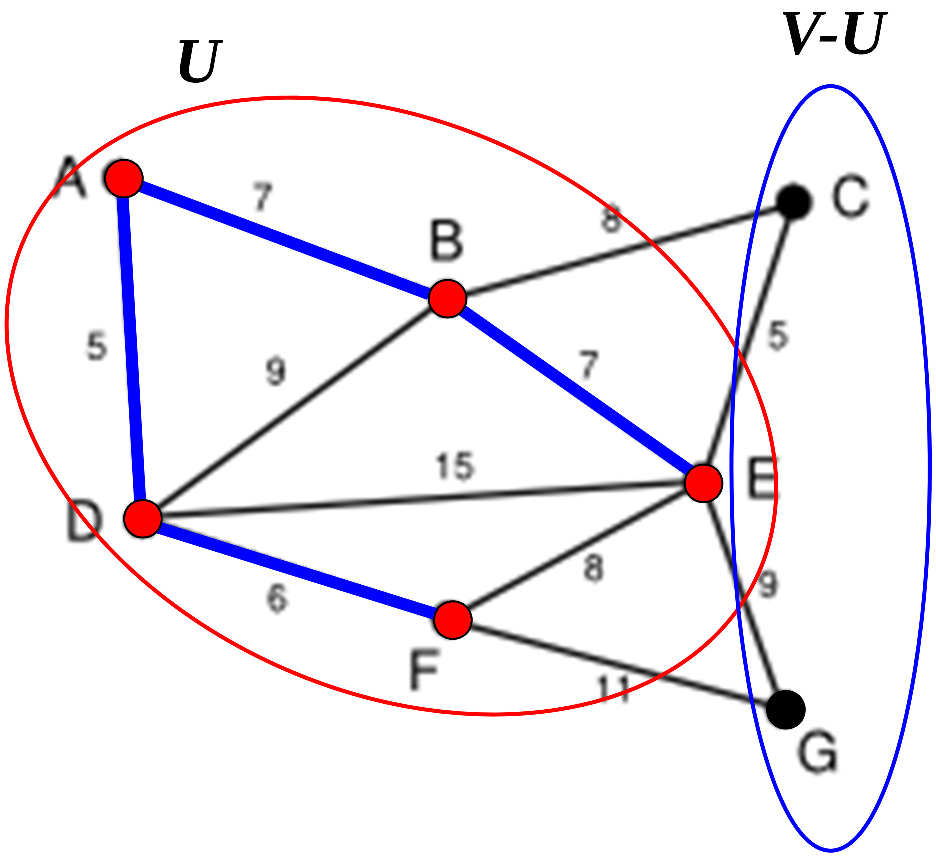
U
A
B
D
F

T
(A,D)
(D,F)
(A,B)

Candidate edges: (B, C): 8
(B, E): 7
(D, E): 15
(F, E): 8
(F, G): 11

3.1.2 Prim's Algorithm

- Ex)

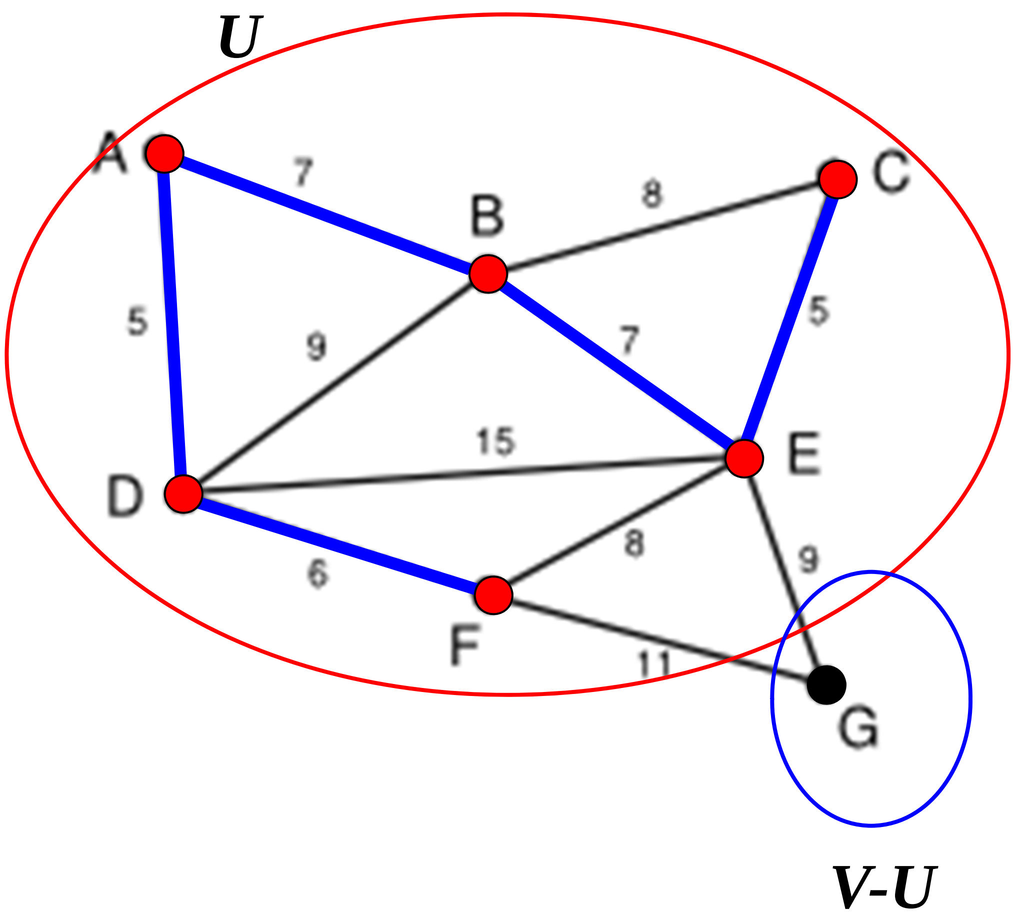


U	***T***
A	(A,D)
B	(D,F)
	(A,B)
D	(B,E)
E	
F	

Candidate edges: (B, C): 8
(E, C): 5
(E, G): 9
(F, G): 11

3.1.2 Prim's Algorithm

- Ex)

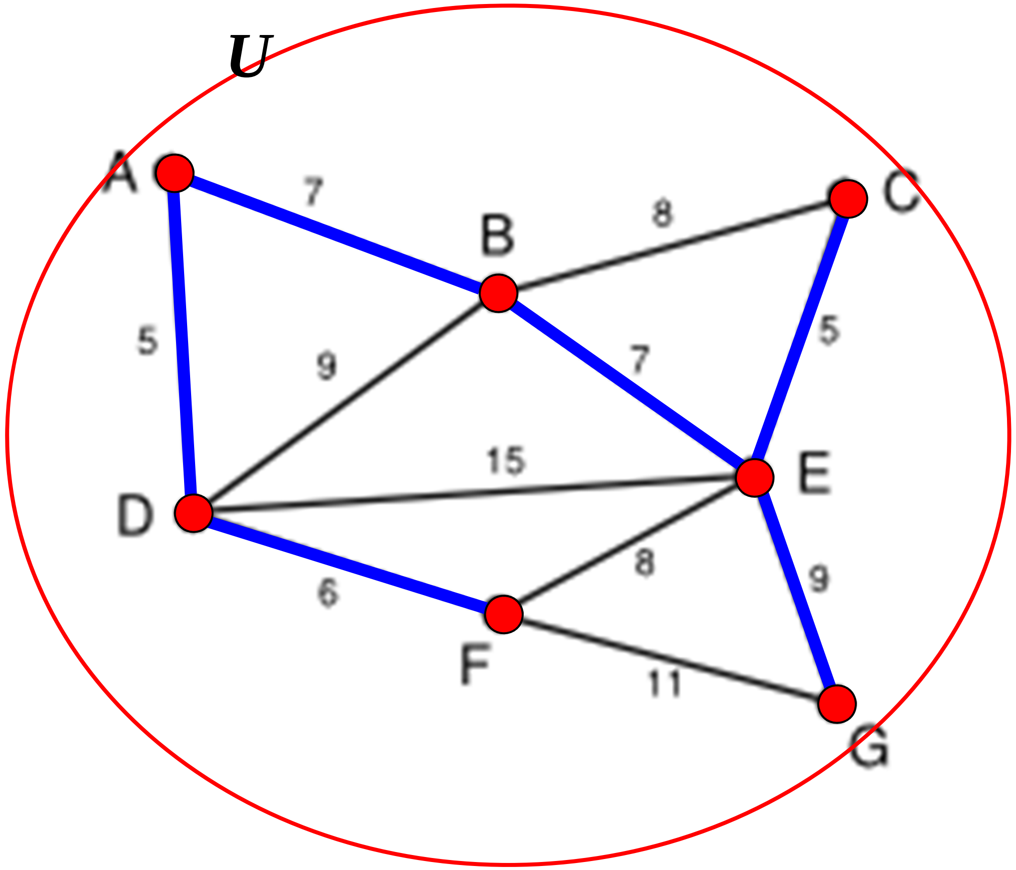


U	T
A	(A,D)
B	(D,F)
C	(A,B)
D	(B,E)
E	(C,E)
F	

Candidate edges: (E, G): 9
(F, G): 11

3.1.2 Prim's Algorithm

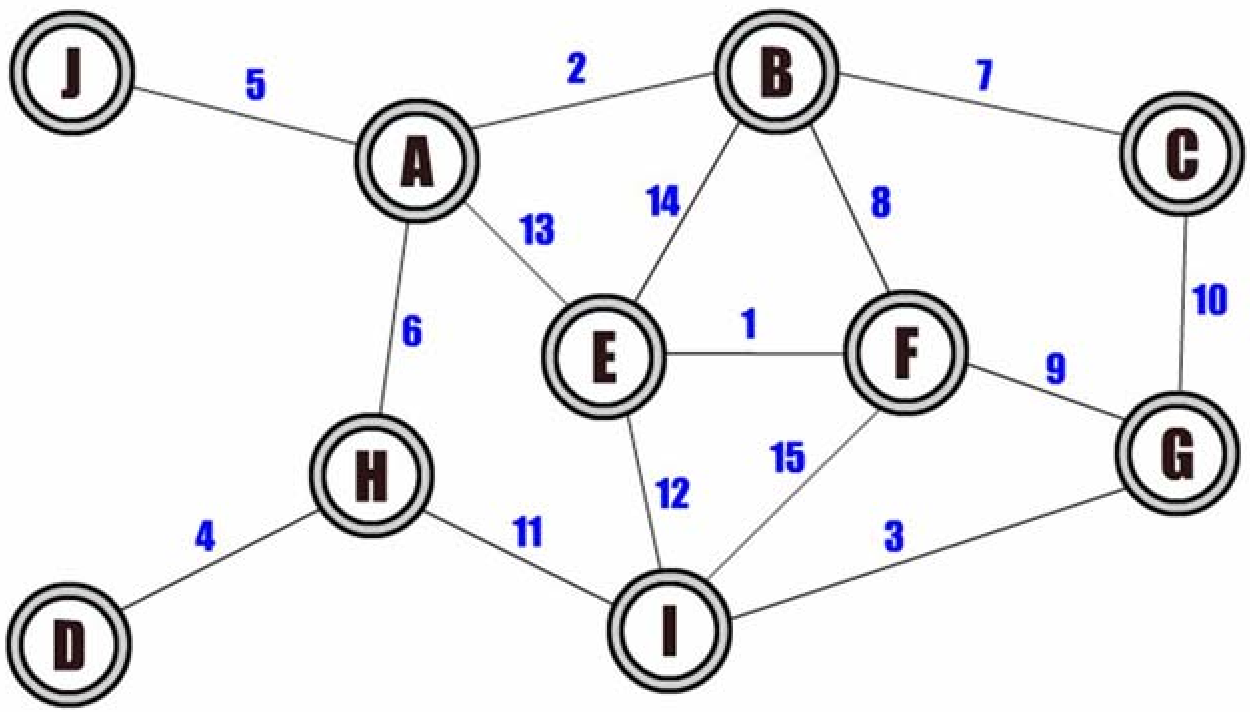
- Ex)



U	*T*
A	(A,D)
B	(D,F)
C	(A,B)
D	(B,E)
E	(C,E)
F	(E,G)
G	

3.1.2 Prim's Algorithm

- Ex)



3.2 Optimal storage on tapes

- Background:
  - There are n files that are to be stored on a computer tape of length L.
  - The length of each file is l_i, $1 \le i \le n$.

3.2 Optimal storage on tapes

- Retrieving files from the tape
  - Initially tape is positioned to the front.
  - The files are stored in the order I = i_1, i_2, …, i_n.
  - The time t_j, needed to retrieve file i_j is

$$t_j = \sum_{1 \le k \le j} l_{i_k}$$

  - MRT (Mean Retrieval Time) $= \frac{1}{n} \sum_{1 \le j \le n} t_j$

3.2 Optimal storage on tapes

- Problem:
  - We are required to find a permutation for the n files so that when they are stored on a tape in this order, the MRT is minium.
  - Minimizing MRT is to minimize D(I)

$$D(I) = \sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{i_k}.$$

3.2 Optimal storage on tapes

- Example:
  - $n = 3$ & $I = (l_1, l_2, l_3) = (5, 10, 3)$
  - We have 6 possible permutations
    - 1, 2, 3
    - 1, 3, 2
    - 2, 1, 3
    - 2, 3, 1
    - 3, 1, 2
    - 3, 2, 1
  - Optimal solution?

3.2 Optimal storage on tapes

- Solution strategy
  - The next file to be stored on the tape would be the one which minimizes the increase of D.
  - If we have already constructed i_1, i_2, …, i_r, and the next file is j, then the following value should be minimum:

$$\sum_{1 \le k \le r} l_{i_k} + l_j$$

3.2 Optimal storage on tapes

- Greedy solution
  - Sort i's in the nondecreasing order of l_i.
  - Store them in the sorted order on the tape.
  - Time complexity: O(n log n).

3.3 Knapsack problem

- Problem:
  - We are given n objects and a knapsack.
  - Object i has a weight w_i and a profit p_i, and the knapsack has a capacity M.
  - If a fraction x_i, $0 \le x_i \le 1$, of object i is placed into the knapsack, then the profit of $p_i x_i$ is earned.

3.3 Knapsack problem

- Problem:
  - The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

$$\text{Maximize} \sum_{1 \le i \le n} p_i x_i \text{ subject to } \sum_{1 \le i \le n} w_i x_i \le M$$

  - The solution is a set $(x_1, x_2, \ldots, x_n)$

3.3 Knapsack problem

- Example
  - $n = 3$, $M = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$, $(w_1, w_2, w_3) = (18, 15, 10)$.
  - What is the solution (x_1, x_2, x_3) ?

3.3 Knapsack problem

- Solution strategy
  - At each step, we include that object which has the maximum profit per unit of capacity.
  - In the order of the ratio p_i / w_i.

3.3 Knapsack problem

- Knapsack algorithm

```
GREEDY_KNAPSACK ( int P[], int W[], int M, int X[], int n )
{
    sort P & W such that P[i]/W[i] >= P[i+1]/W[i+1];
    X ← NULL;
    cu ← M;          //     the remaining knapsack capacity
    for ( i = 0 to n )
        if ( W[i] > cu )
            break;
        X[i] = 1;
        cu ← cu - W[i];
    if ( i <= n )
        X[i] ← cu / W[i];
}
```

3.4 Job sequencing with deadline

- Problem:
  - We are given n jobs.
  - Each job i has a deadline $d_i \geq 0$ and a profit $p_i \geq 0$.
  - For any job i, the profit p_i is earned if and only if the job is completed by its deadline.

3.4 Job sequencing with deadline

- Problem:
  - In order to complete a job, one has to process the job on a machine for one unit of time.
  - Feasible solution for this problem is a subset, J, of jobs such that each job in this subset can be completed by its deadline.
  - The value of J is $\Sigma_i p_i$.
  - Optimal solution: J with maximum value

3.4 Job sequencing with deadline

- Example:
  - $n = 4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$.
  - What are the feasible solutions and their values?

3.4 Job sequencing with deadline

- Solution strategy
  - J is a set of k jobs and $\sigma = i_1, i_2, \ldots, i_k$ be a permutation of jobs such that $d_{i1} \leq d_{i2} \leq \ldots \leq d_{ik}$.
  - J is a feasible solution if and only if the jobs in J can be processed in the order σ without violating any deadline.
  - $D(J(1)) \leq D(J(2)) \leq \ldots \leq D(J(k))$.
  - $D(J(r)) \geq r$, for $1 \leq r \leq k$.

3.4 Job sequencing with deadline

- Solution strategy
  - We assume that the jobs are sorted such that $p_1 \geq p_2 \geq \ldots \geq p_n$.
  - We assume that min { D(i) } = 1.

  - **Select the job in the non-ascending order of profit.**
  - **If the pre-selected jobs can yield, then make them yield as much as possible.**

3.4 Job sequencing with deadline

- Job scheduling

```
void JobSchedule( int D[], int J[], int n )
//  initially jobs are sorted such that p1 ≥ p2 ≥ … ≥ pn
{
    D[0] ← J[0] ← 0;
    k ← 1; J[1] ← 1;
    for ( i ← 2 to n by 1 )
        r ← k;
        while ( D[J[r]] > D[i] and D[J[r]] != r )
            r ← r - 1;
        if ( D[J[r]] <= D[i] and D[i] > r )
            for ( l = k; l >= r + 1 by -1 )
                J[l+1] ← J[l];
            J[r+1] ← i; k ← k+1;
}
```

3.4 Job sequencing with deadline

- Job scheduling

```
void JobSchedule( int D[], int J[], int n )
//  initially jobs are sorted such that p1 ≥ p2 ≥ … ≥ pn
{
    D[0] ← J[0] ← 0;
    k ← 1; J[1] ← 1;
    for ( i ← 2 to n by 1 )
        r ← k;
        while ( D[J[r]] > D[i] and D[J[r]] != r )
            r ← r - 1;
        if ( D[J[r]] <= D[i] and D[i] > r )
            for ( l = k; l >= r + 1 by -1 )
                J[l+1] ← J[l];
            J[r+1] ← i; k ← k+1;
}
```

Find the one that can yield

3.4 Job sequencing with deadline

- Job scheduling

```
void JobSchedule( int D[], int J[], int n )
//  initially jobs are sorted such that p₁ ≥ p₂ ≥ … ≥ pₙ
{
    D[0] ← J[0] ← 0;
    k ← 1; J[1] ← 1;
    for ( i ← 2 to n by 1 )
        r ← k;
        while ( D[J[r]] > D[i] and D[J[r]] != r )
            r ← r - 1;
        if ( D[J[r]] <= D[i] and D[i] > r )
            for ( l = k; l >= r + 1 by -1 )
                J[l+1] ← J[l];
            J[r+1] ← i; k ← k+1;
}
```

If feasible, insert the job

3.4 Job sequencing with deadline

- Example:
  - $n = 5$, $(p_1, p_2, p_3, p_4, p_5) = (20, 15, 10, 5, 1)$, $(d_1, d_2, d_3, d_4, d_5) = (3, 4, 1, 2, 5)$

Condition of feasibility?

$D(J(1)) \le D(J(2)) \le \ldots \le D(J(k))$

&

$D(J(r)) \ge r$

3.4 Job sequencing with deadline

- Example:
  - $n = 5$,

    $(p_1, p_2, p_3, p_4, p_5) = (\boxed{20,}\ 15, 10, 5, 1)$,
    $(d_1, d_2, d_3, d_4, d_5) = (\boxed{3,}\ 4, 1, 2, 5)$

r		**1**	**2**	**3**	**4**	**5**
J(r)	**0**	**1**				

feasible

D(J(r))	**0**	**D(J(1)) =3**				

3.4 Job sequencing with deadline

- Example:
  - $n = 5$,

    $(p_1, p_2, p_3, p_4, p_5) = (20, 15, 10, 5, 1)$,
    $(d_1, d_2, d_3, d_4, d_5) = (3, 4, 1, 2, 5)$

r		1	2	3	4	5
J(r)	**0**	**1**	**2**			

feasible

D(J(r))	**0**	**D(J(1)) =3**	**D(J(2)) =4**			

3.4 Job sequencing with deadline

- Example:
  - $n = 5$,

    $(p_1, p_2, p_3, p_4, p_5) = (20, 15, \boxed{10}, 5, 1)$,

    $(d_1, d_2, d_3, d_4, d_5) = (3, 4, \boxed{1}, 2, 5)$

r		1	2	3	4	5
J(r)	**0**	**1**	**2**	**3**		

Not feasible

D(J(r))	**0**	**D(J(1)) = 3**	**D(J(2)) = 4**	**D(J(3)) = 1**		

3.4 Job sequencing with deadline

- Example:
  - $n = 5$,

    $(p_1, p_2, p_3, p_4, p_5) = (20, 15, \boxed{10}, 5, 1)$,
    $(d_1, d_2, d_3, d_4, d_5) = (3, 4, \boxed{1}, 2, 5)$

r		1	2	3	4	5
J(r)	**0**	**3**	**1**	**2**		

feasible

D(J(r))	**0**	**D(J(1)) = 1**	**D(J(2)) = 3**	**D(J(3)) = 4**		

3.4 Job sequencing with deadline

- Example:
  - $n = 5$,

    $(p_1, p_2, p_3, p_4, p_5) = (20, 15, 10, \boxed{5,}\ 1)$,
    $(d_1, d_2, d_3, d_4, d_5) = (3, 4, 1, \boxed{2,}\ 5)$

r		1	2	3	4	5
J(r)	**0**	**3**	**1**	**2**	**4**	
					Not feasible	
D(J(r))	**0**	**D(J(1)) = 1**	**D(J(2)) = 3**	**D(J(3)) = 4**	**D(J(4))) = 2**	

3.4 Job sequencing with deadline

- Example:
  - $n = 5$,

    $(p_1, p_2, p_3, p_4, p_5) = (20, 15, 10, 5, 1)$,
    $(d_1, d_2, d_3, d_4, d_5) = (3, 4, 1, 2, 5)$

r		1	2	3	4	5
J(r)	0	3	4	1	2	
					feasible	
D(J(r))	0	D(J(1)) = 1	D(J(2)) = 2	D(J(3)) = 3	D(J(4))) = 4	

3.4 Job sequencing with deadline

- Example:
  - $n = 5$,

    $(p_1, p_2, p_3, p_4, p_5) = (20, 15, 10, 5, 1)$,
    $(d_1, d_2, d_3, d_4, d_5) = (3, 4, 1, 2, 5)$

r		1	2	3	4	5
J(r)	0	3	4	1	2	5
						feasible
D(J(r))	0	D(J(1)) = 1	D(J(2)) = 2	D(J(3)) = 3	D(J(4))) = 4	D(J(5))) = 5

3.5 Optimal merge pattern

- Problem:
  - Merge k files → Various combinations of merging patterns
  - c.f. Merging two sorted files containing n and m records into one file takes O(n+m).
  - Determine an optimal way to pairwisely merge k sorted files together.

3.5 Optimal merge pattern

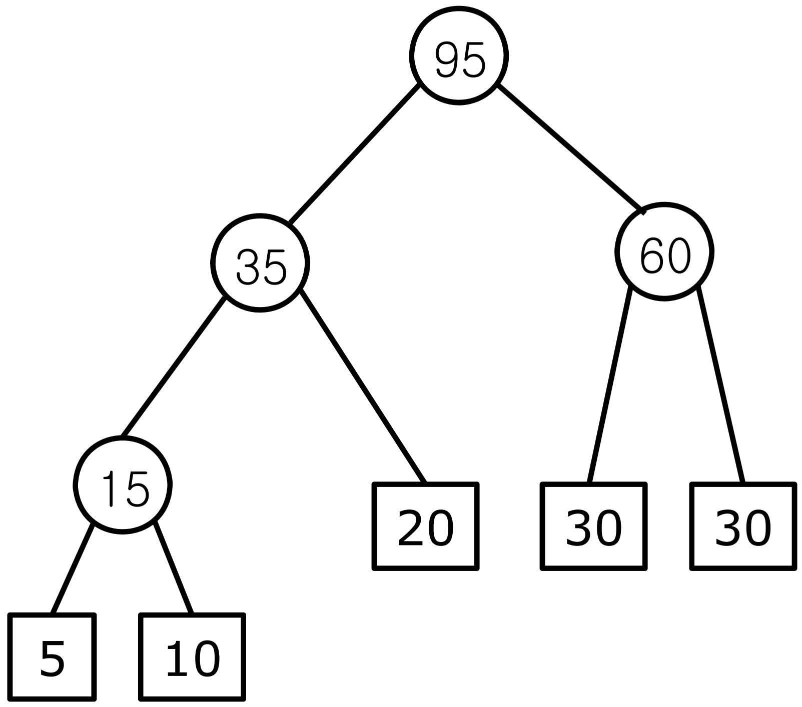
- Example:
  - X1, X2 and X3 are three sorted files of length 30, 20, and 10 records each.
  - Merge pattern 1:
    - Merge X1 & X2 → Y1 (50 steps)
    - Merge Y1 & X3 → Y2 (60 steps)
  - Merge pattern 2:
    - Merge X2 & X3 → Y1 (30 steps)
    - Merge Y1 & X1 → Y2 (60 steps)
  - Compare the time required

3.5 Optimal merge pattern

- Solution strategy:
  - 2-way merge pattern can be represented as a binary merge tree with minimum weighted external path length
  - Example:
    - Five patterns with (20, 30, 10, 5, 30)

3.5 Optimal merge pattern

- Solution strategy:
  - Corresponding binary tree



3.5 Optimal merge pattern

- Solution strategy:
  - If a pattern of length q_i is stored at a node whose depth is d_i, then the number of moves of the pattern is $q_i\ d_i$.
  - The total moves of the patterns is:

$$\sum_{1 \le i \le n} d_i q_i$$

  - The weighted external path length of a tree

3.5 Optimal merge pattern

- Optimal merge pattern

```
int build_tree ( node *tree, int n, int L[] )
{
    for ( i = 1 to n-1 ) {
        tnode ← build_a_node ( );
        tnode->left ← least ( L );
        tnode->right ← least ( L );
        tnode->weight ← tnode->left->weight
                      + tnode->right->weight;
        insert ( tnode, tree );
    }

    return tree->weight;
}
```

3.6 Huffman encoding

- Example: The encoding process of MP3
  - Sample at regular rates
    - In CD, 44,100 samples per second
    - A 50 min-length symphony has 50 X 60 X 44,100 ≈ 130,000,000 samples
  - Each sample is quantized
    - Each sample value is approximated by a nearby number from a finite set T.
  - The resulting string is encoded in binary

3.6 Huffman encoding

- Size of encoding
  - Estimating the size of encoding
    - Number of samples X T.
    - If T = {A, B, C, D}, previous example has 130,000,000 X 2 bits = 37.5MByte
    - If T has M symbols, encoding for T requires ($\log_2 M$) bits.
  - How can we reduce the size of encoding?
    - Reduce sample rates
    - Reduce the length of alphabets in T

3.6 Huffman encoding

- Reduce the length of alphabets in T
  - Greedy approach

    → Variable-length encoding

Alphabet	Frequency	Conventional	Variable-length
A	70M	00	0
B	3M	01	001
C	20M	10	10
D	37M	11	11

- The resulting size of encoding becomes

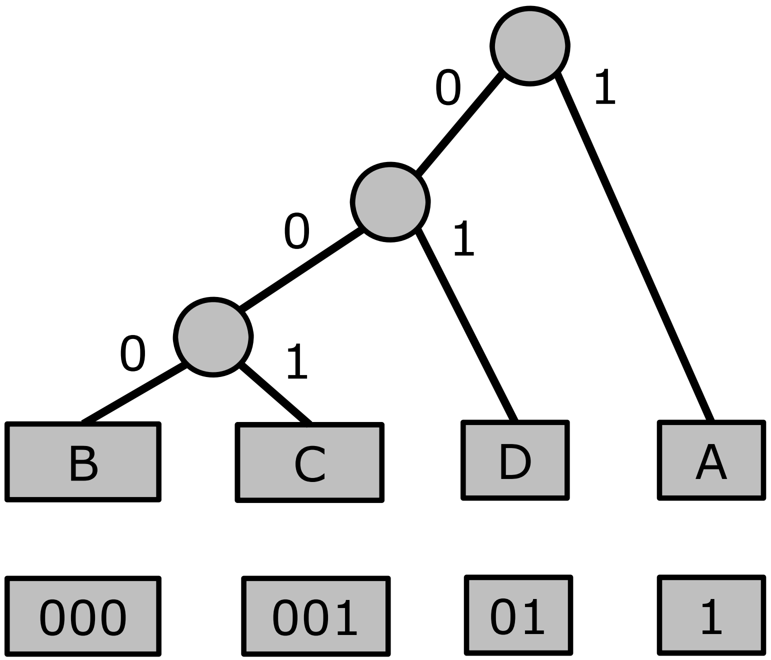
Type	Formula	Size
Conventional	130,000,000 X 2 bits	260M bits
Variable-length	70M X 1 bit + 57M X 2 bits + 3M X 3bits	193M bits

3.6 Huffman encoding

- Problem of previous encoding?
  - Ambiguity
  - What is 0010?
    - AAC or BA
  - How to avoid ambiguity?
    - Prefix-free encoding
      - No codeword can be a prefix of another codeword
      - Can be represented using full binary tree
- Huffmann encoding
  - Variable-length & prefix-free encoding
  - Similar to optimal merge pattern algorithm

3.6 Huffman encoding

- Strategy
  - Similar to optimal merge pattern
  - Sort the symbols according to the increasing order of frequency



3.6 Huffman encoding

- Huffmann encoding
  - Variable-length & prefix-free encoding

Alphabet	Frequency	Conventional	Variable-length	Huffman code
A	70M	00	0	1
B	3M	01	001	000
C	20M	10	10	001
D	37M	11	11	01

Type	Formula	Size
Conventional	130,000,000 X 2 bits	260M bits
Variable-length	70M X 1 bit + 57M X 2 bits + 3M X 3bits	193M bits
Huffman code	70M X 1 bit + 37M X 2 bits + 23M X 3bits	213M bits

3. Greedy algorithm

3.0 Basics

3.1 Minimum spanning trees

3.2 Optimal storage on tapes

3.3 Knapsack problem

3.4 Job sequencing with deadline

3.5 Optimal merge patterns

3.6 Huffman encoding

Contents

0. Prologue

1. Divide & conquer

2. Graph

3. Greedy algorithm

4. Dynamic programming