Chapter 9 Trees 树

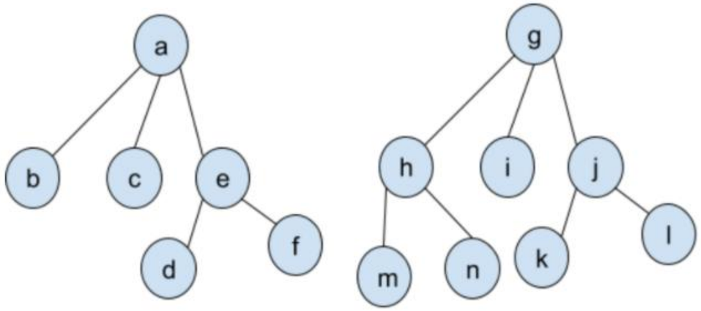
Lu Han

hl@bupt.edu.cn

9.2 Terminology and Characterizations of Trees 树的术语和性质

If T is a graph with n vertices, the following are equivalent (Theorem 9.2.3):
(a) T is a tree.
(b) If v and w are vertices in T, there is a unique simple path from v to w.
(c) T is connected and acyclic.
(d) T is connected and has $n - 1$ edges.
(e) T is acyclic and has $n - 1$ edges.

A **forest (森林)** is a simple graph with no cycles.

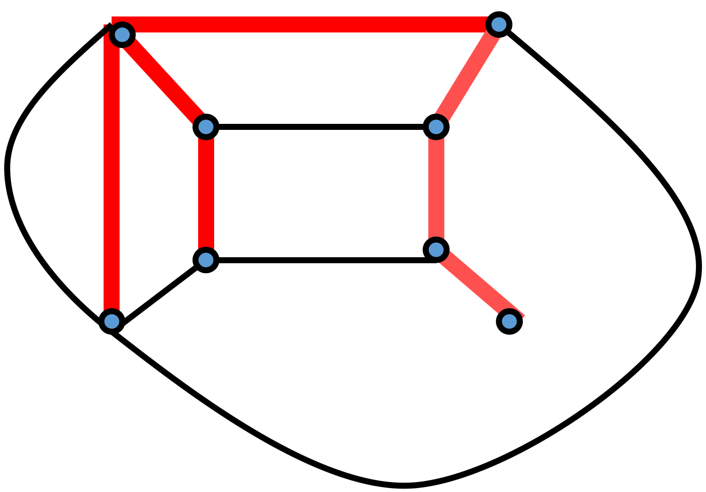


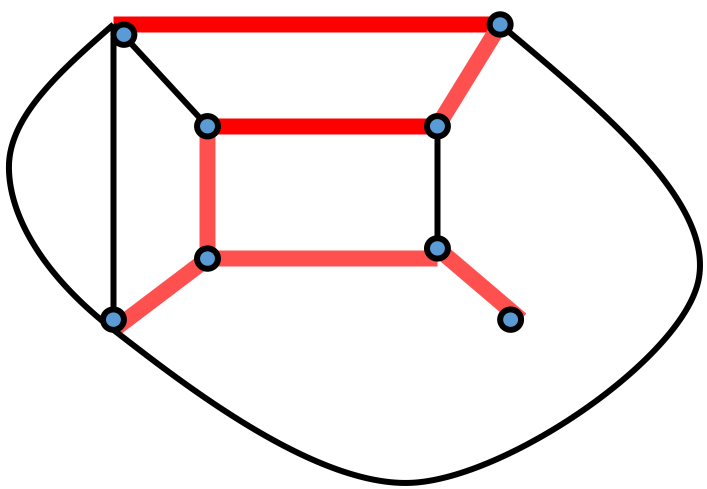
Forest

9.3 Spanning Trees 生成树

Definition 9.3.1 A tree T is a **spanning tree (生成树)** of a graph G if T is a subgraph of G that contains all of the vertices of G.

In general, a graph will have several spanning trees.



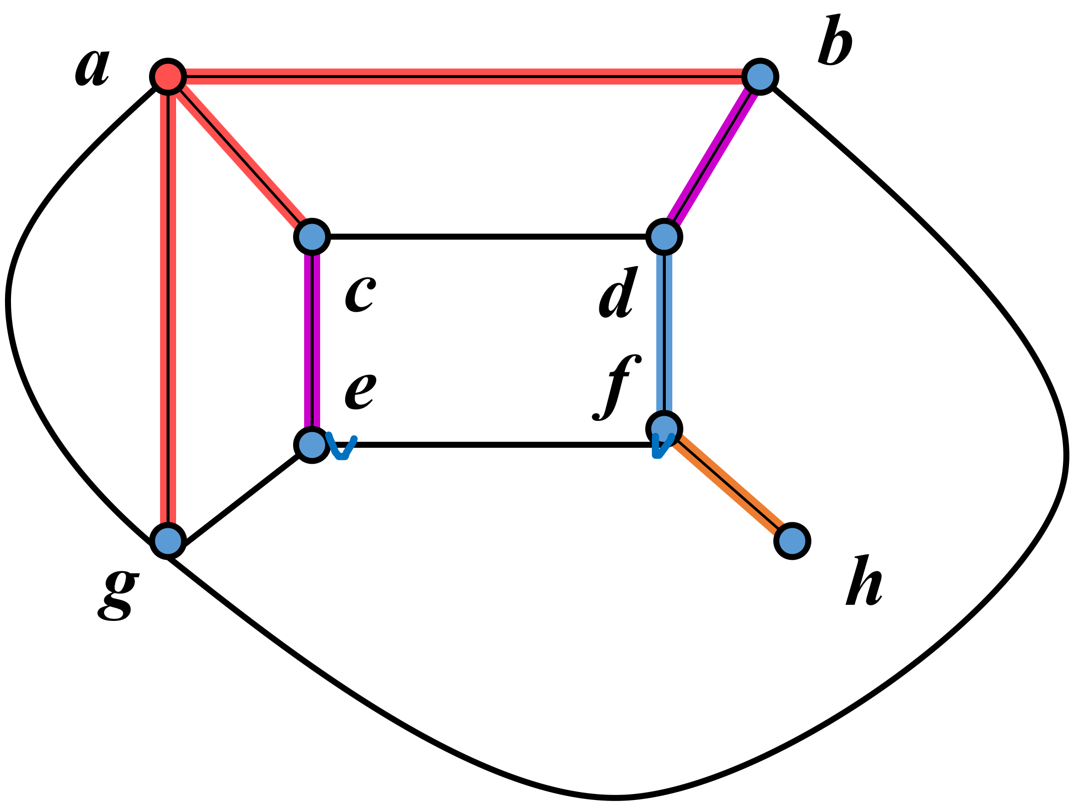


9.3 Spanning Trees 生成树

Theorem 9.3.4 A graph G has a spanning tree if and only if G is connected.

9.3 Spanning Trees 生成树

Breadth-First Search 广度优先搜索



- Select an ordering, say $abcdefgh$, of the vertices of G.
- Select the first vertex a and label it the root. Let T consist of the single vertex a and no edges.
- Add to T all edges (a, x) and vertices on which they are incident, for $x = b$ to h, that do not produce a cycle when added to T.
- Repeat this procedure with the vertices on level 1 (2, 3, ...) by examing each in order.
- Since no edge can be added to the single vertex h on lever 4, the procedure ends.

9.3 Spanning Trees 生成树

Breadth-First Search 广度优先搜索

Input: A connected graph G with vertices ordered

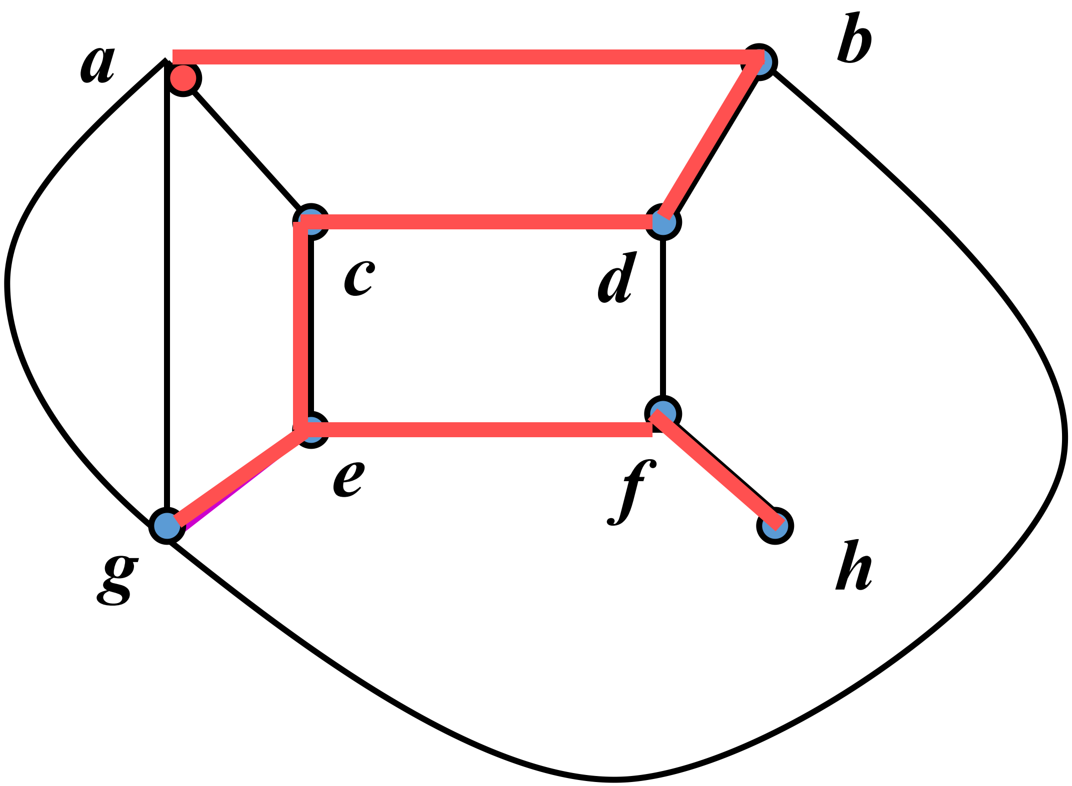
$$v_1, v_2, \ldots, v_n$$

Output: A spanning tree T

```
bfs(V, E) {
    // V = vertices ordered v1, ..., vn; E = edges
    // V′ = vertices of spanning tree T; E′ = edges of spanning tree T
    // v1 is the root of the spanning tree
    // S is an ordered list
    S = (v1)
    V′ = {v1}
    E′ = ∅
    while (true) {
        for each x ∈ S, in order,
            for each y ∈ V − V′, in order,
                if ((x, y) is an edge)
                    add edge (x, y) to E′ and y to V′
        if (no edges were added)
            return T
        S = children of S ordered consistently with the original vertex ordering
    }
}
```

9.3 Spanning Trees 生成树

Depth-First Search 深度优先搜索



- Select an ordering, say $abcdefgh$, of the vertices of G.
- Select the first vertex a and label it the root. Let T consist of the single vertex a and no edges.
- Add to T the edge (a, x) with minimal x and the vertex x, which is incident and does not produce a cycle when added to T.
- Repeat this procedure with the vertex on the next level until we cannot add an edge.
- Backtrack to the parent of the current vertx and try to add an edge.
- When no more edges can be added, we finally backtrack to the root and algorithm ends.

9.3 Spanning Trees 生成树

Depth-First Search 深度优先搜索

Input: A connected graph G with vertices ordered

$$v_1, v_2, \ldots, v_n$$

Output: A spanning tree T

```
dfs(V, E) {
    // V' = vertices of spanning tree T; E' = edges of spanning tree T
    // v1 is the root of the spanning tree
    V' = {v1}
    E' = ∅
    w = v1
    while (true) {
        while (there is an edge (w, v) that when added to T does not create a cycle
            in T) {
            choose the edge (w, vk) with minimum k that when added to T
                does not create a cycle in T
            add (w, vk) to E'
            add vk to V'
            w = vk
        }
        if (w == v1)
            return T
        w = parent of w in T // backtrack
    }
}
```

9.3 Spanning Trees 生成树

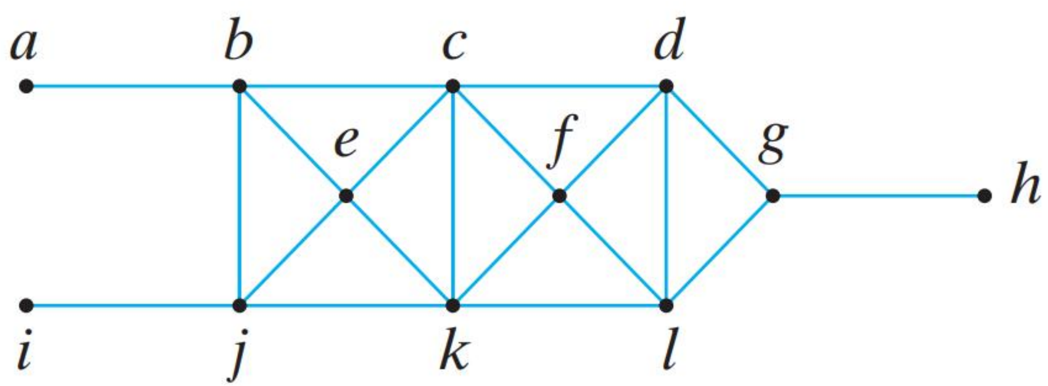
Breadth-First Search 广度优先搜索

The idea of breadth-first search is to process all the vertices on a given level before moving to next-higher level.

Depth-First Search 深度优先搜索 → Backtracking 回溯

The idea of depth-first search is to proceeds to successive levels in a tree at the earliest possible opportunity.

Exercise Find a spanning tree for the graph using Breadth-First Search.



9.3 Spanning Trees 生成树

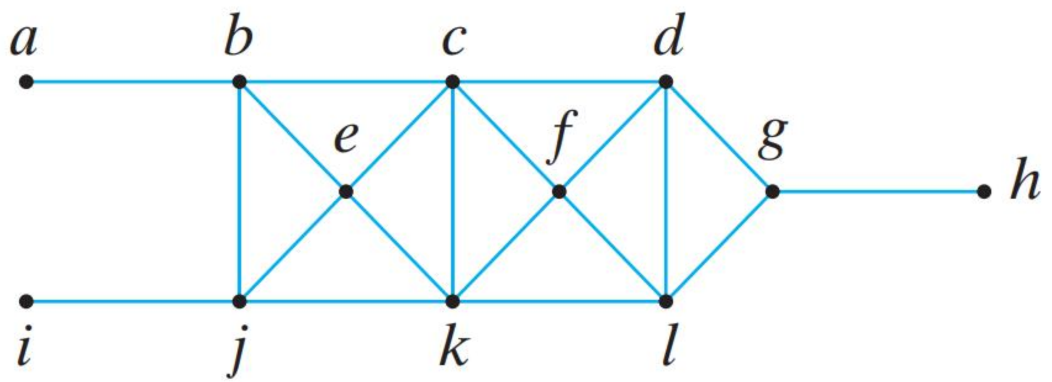
Breadth-First Search 广度优先搜索

The idea of breadth-first search is to process all the vertices on a given level before moving to next-higher level.

Depth-First Search 深度优先搜索 → Backtracking 回溯

The idea of depth-first search is to proceeds to successive levels in a tree at the earliest possible opportunity.

Exercise Find a spanning tree for the graph using Depth-First Search.

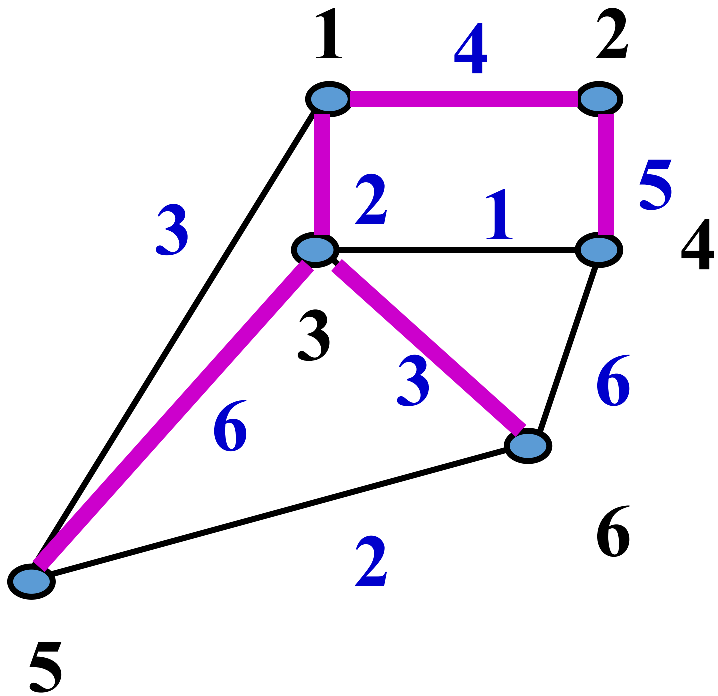


9.4 Minimal Spanning Trees 最小生成树

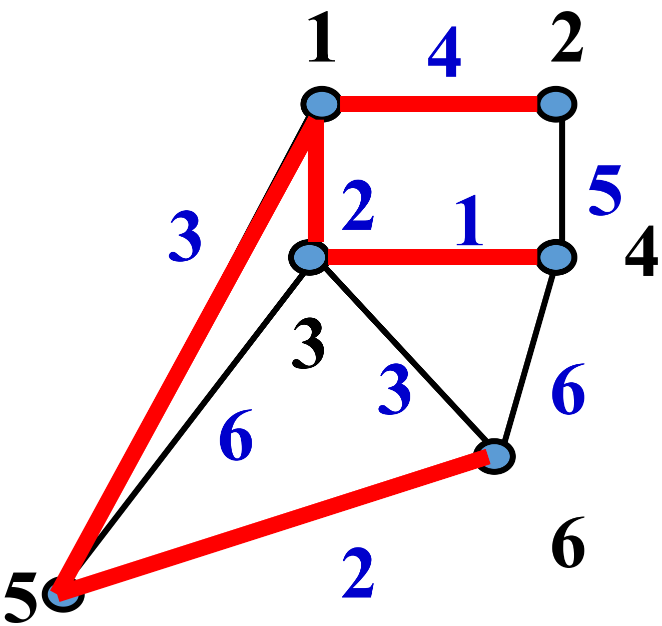
Definition 9.4.1 Given a weighted graph G, a **minimal spanning tree (最小生成树)** of G is a spanning tree of G that has minimum weight.

9.4 Minimal Spanning Trees 最小生成树

Definition 9.4.1 Given a weighted graph G, a **minimal spanning tree (最小生成树)** of G is a spanning tree of G that has minimum weight.



Weight=20

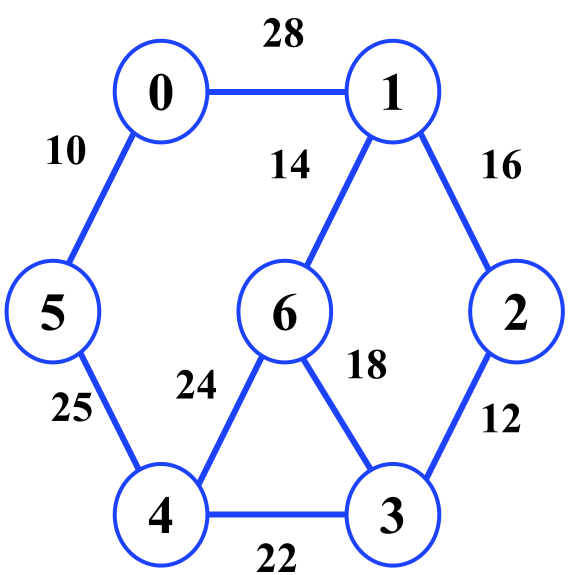


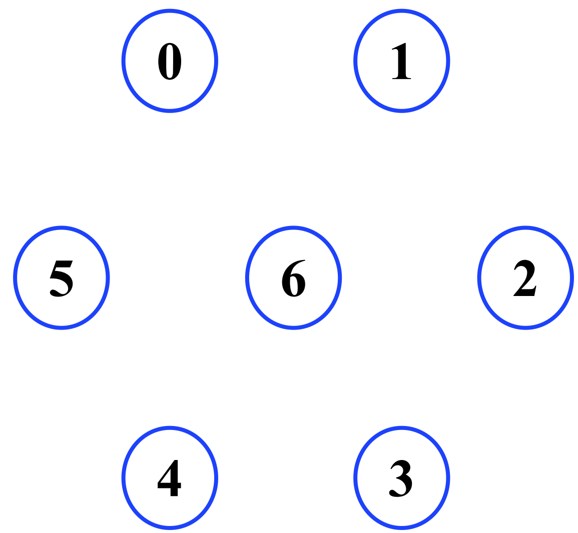
Weight=12

9.4 Minimal Spanning Trees 最小生成树

Prim’s Algorithm 普里姆算法

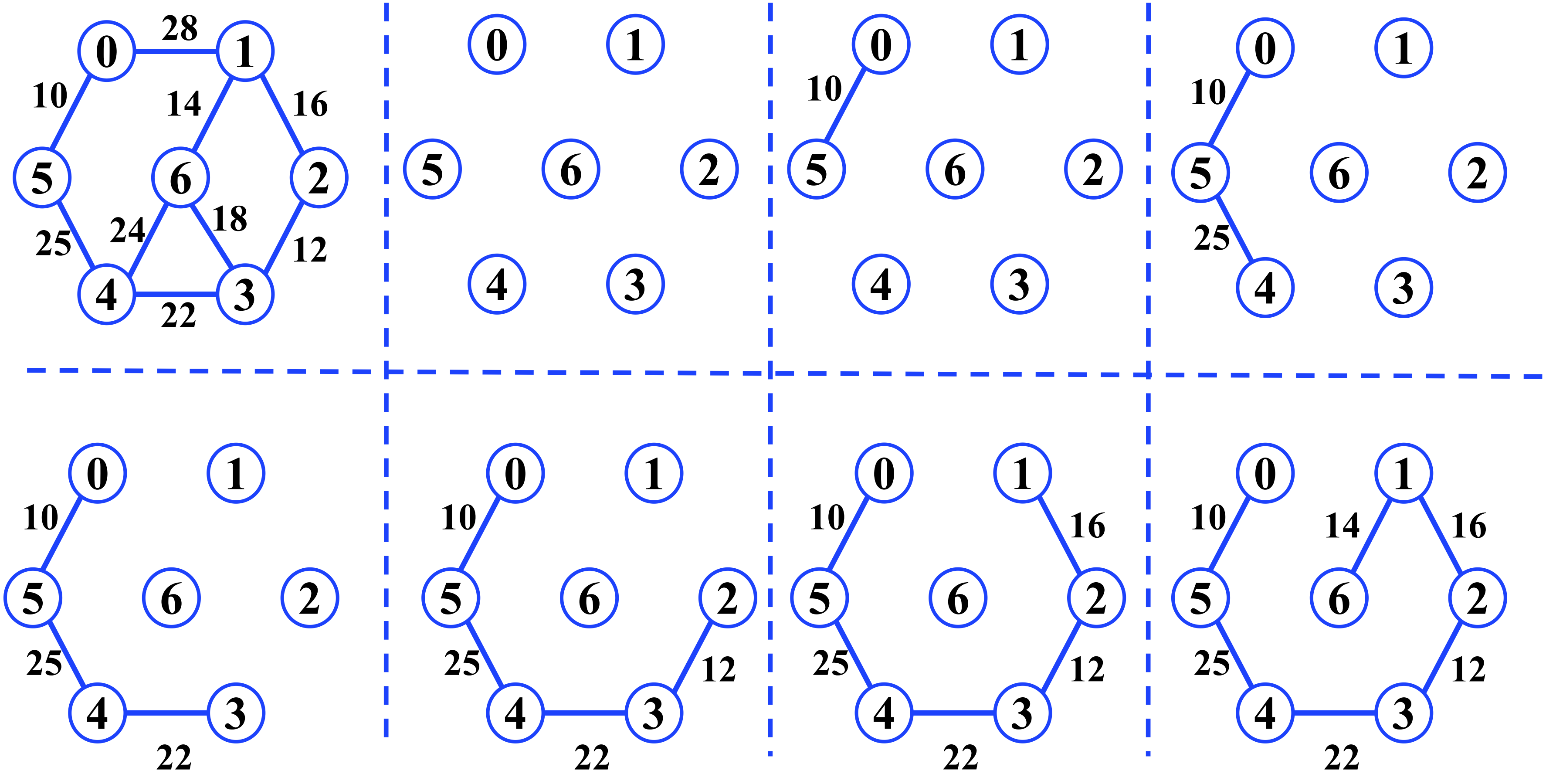
The algorithm begins with a single vertex. Then at each iteration, it adds to the current tree a minimum-weight edge that does not complete a cycle.





Keep finding the minimum-weight edge with one vertex in the tree and one vertex not in the tree.

Prim's Algorithm 普里姆算法



9.4 Minimal Spanning Trees 最小生成树

Prim's Algorithm 普里姆算法

The algorithm begins with a single vertex. Then at each iteration, it adds to the current tree a minimum-weight edge that does not complete a cycle.

Step 0: Pick any vertex as a starting vertex (call it a). $T = \{a\}$.

Step 1: Find the edge with smallest weight incident to a. Add it to T. Also include in T the next vertex and call it b.

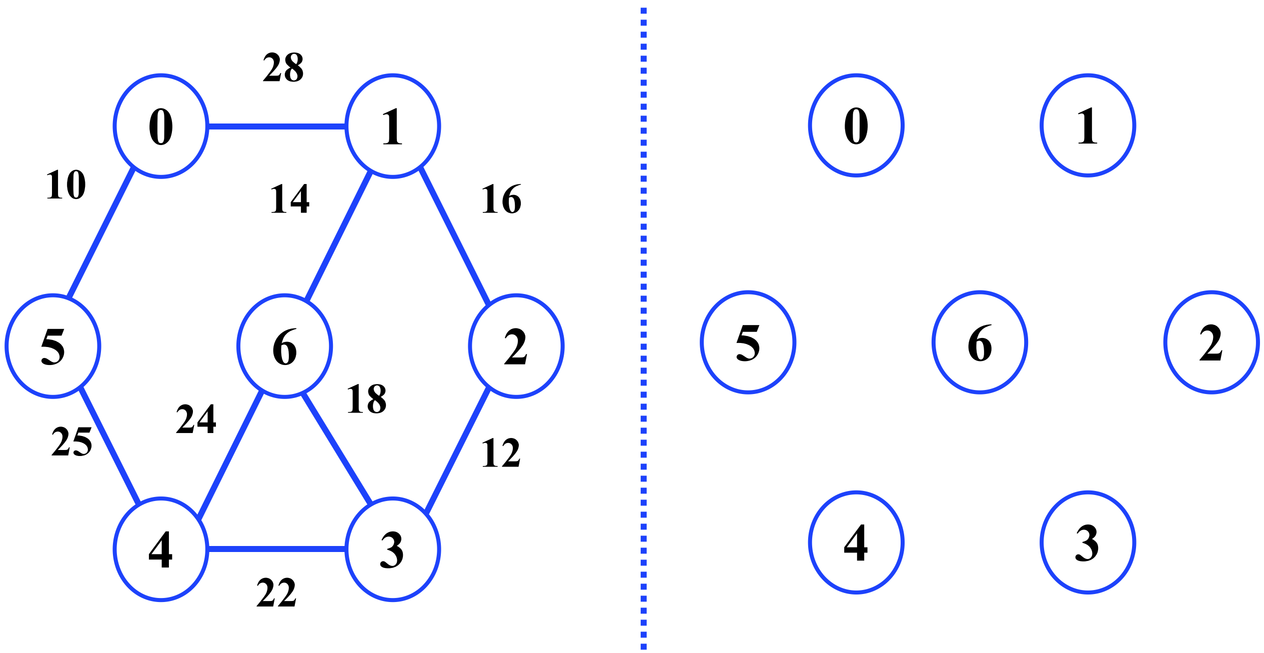
Step 2: Find the edge of smallest weight incident to either a or b. Include in T that edge and the next incident vertex. Call that vertex c.

Step 3: Repeat Step 2, choosing the edge of smallest weight that does not form a cycle until all vertices are in T. The resulting subgraph T is a minimum spanning tree.

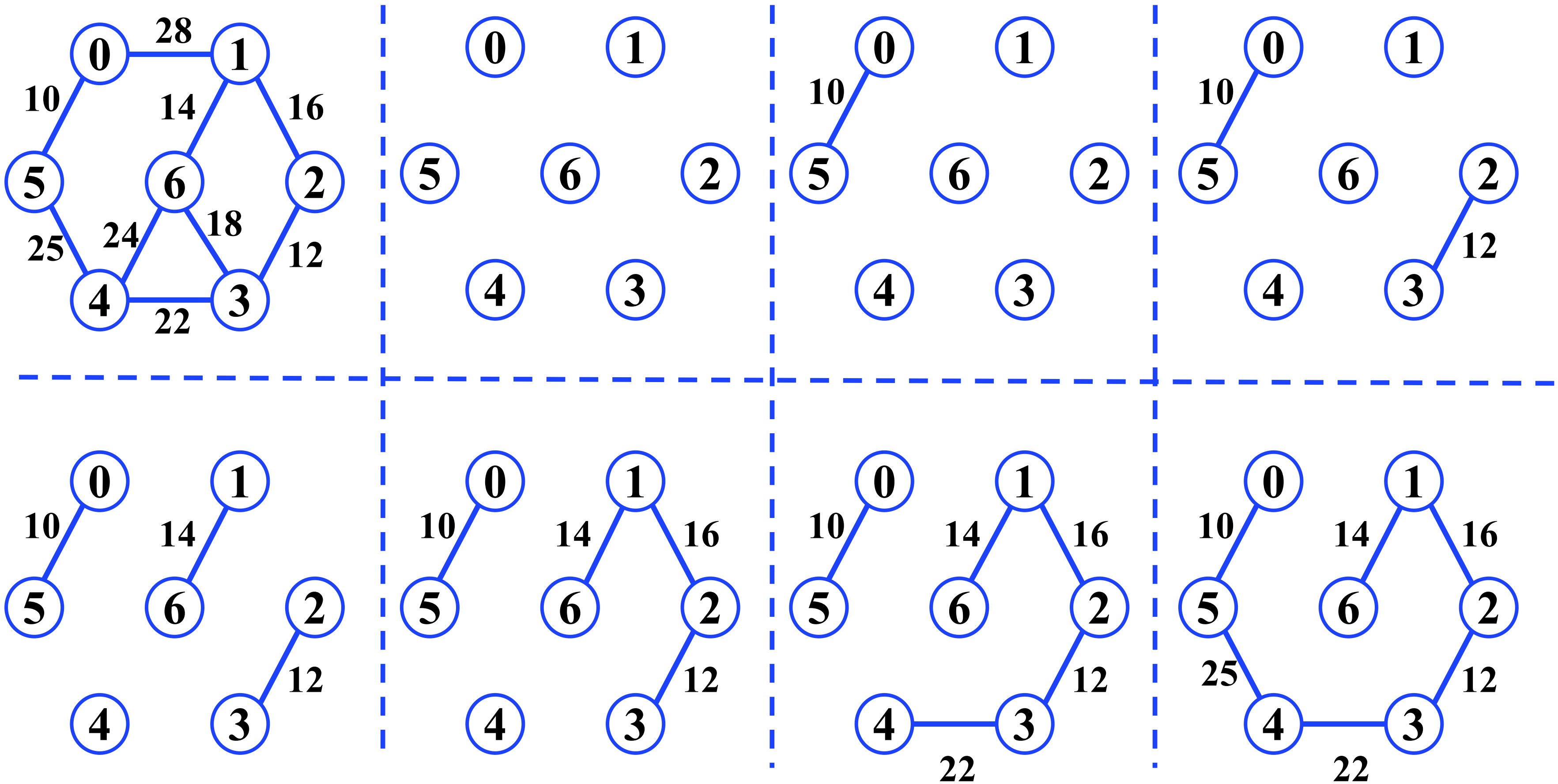
9.4 Minimal Spanning Trees 最小生成树

Kruskal's Algorithm 克鲁斯卡尔算法

It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest.



Kruskal's Algorithm 克鲁斯卡尔算法



9.4 Minimal Spanning Trees 最小生成树

Kruskal's Algorithm 克鲁斯卡尔算法
It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest.

Step 1: Find the edge in the graph with smallest weight (if there is more than one, pick one at random).

Step 2: Find the next edge in the graph with smallest weight that doesn't close a cycle.

Step 3: Repeat Step 2 until you reach out to every vertex of the graph. The chosen edges form the desired minimum spanning tree.

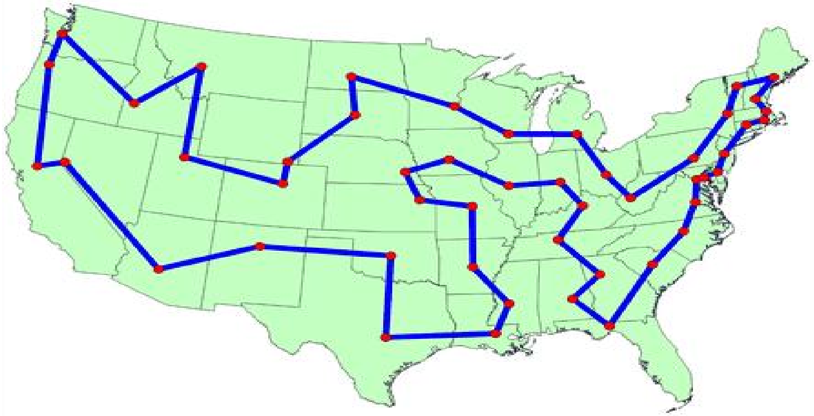
Minimal Spanning Trees and Metric TSP

Traveling salesman problem (TSP)

Given a complete graph with nonnegative edge costs, find a minimum cost cycle visiting every vertex exactly once.

Metric TSP

The edge costs satisfy triangle inequality.



Minimal Spanning Trees and Metric TSP

Traveling salesman problem (TSP)
Given a complete graph with nonnegative edge costs, find a minimum cost cycle visiting every vertex exactly once.

Metric TSP
The edge costs satisfy triangle inequality.

The Metric TSP is an NP-hard problem.
If $\mathrm{P} \neq \mathrm{NP}$, we can't simultaneously have algorithms that
(1) find optimal solutions
(2) in polynomial-time
(3) for any instance.
At least one of these requirements must be relaxed in any approach to dealing with an NP-hard optimization problem.

P & NP

P：(Decision) problems solvable in Polynomial time

P & NP

P： (Decision) problems solvable in Polynomial time

Why polynomial time is important?

$O(logn)$ $O(n)$ $O(n\log n)$ $O(n^c)$ $O(2^n)$ $O(n!)$

Time complexity

P & NP

P： (Decision) problems solvable in Polynomial time

Why polynomial time is important?

$$\overbrace{O(logn) \quad O(n) \quad O(nlogn) \quad O(n^c)}^{\text{Polynomial Time}} \quad \Big| \quad O(2^n) \quad O(n!)$$

Time complexity

P & NP

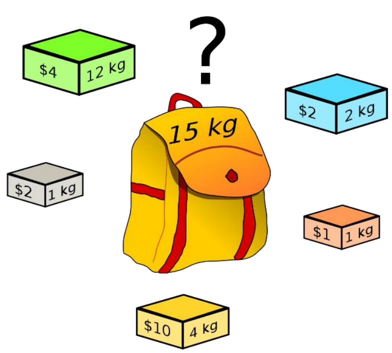
P：(Decision) problems solvable in Polynomial time

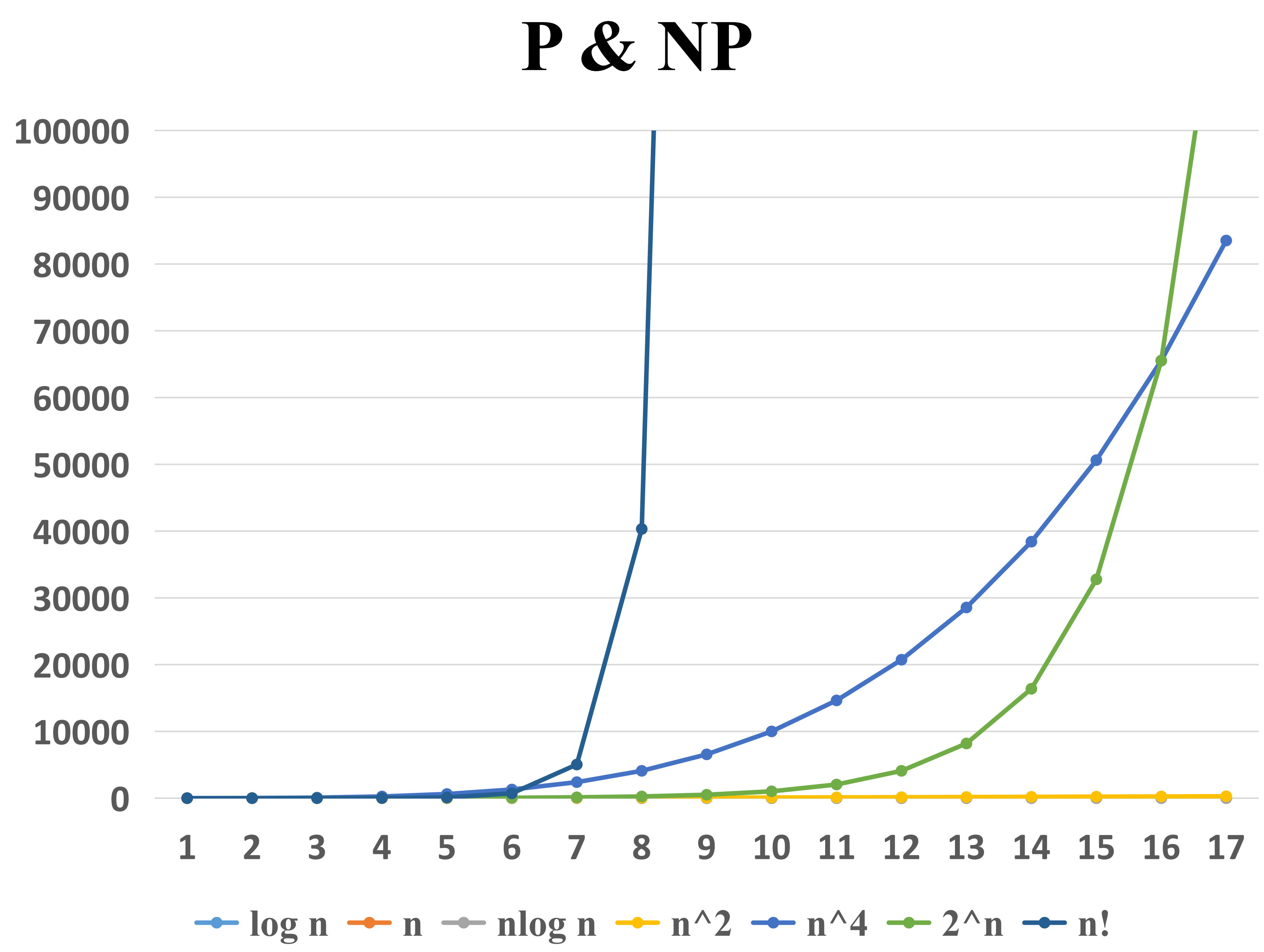
Why polynomial time is important?

Polynomial Time

$\underbrace{O(\log n) \quad O(n) \quad O(n\log n) \quad O(n^c)}_{}$ | $O(2^n) \quad O(n!)$

Time complexity





P & NP
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
log n
n
nlog n
n^2
n^4
2^n
n!

P & NP

	$f(n)$	$n = 20$	$n = 40$	$n = 60$
算法 1	$\log_2 n$	4.32×10^{-6}秒	5.32×10^{-6}秒	5.91×10^{-6}秒
算法 2	$\sqrt{n}$	4.47×10^{-6}秒	6.32×10^{-6}秒	7.75×10^{-6}秒
算法 3	n	20×10^{-6}秒	40×10^{-6}秒	60×10^{-6}秒
算法 4	$n \log_2 n$	86×10^{-6}秒	213×10^{-6}秒	354×10^{-6}秒
算法 5	n^2	400×10^{-6}秒	1600×10^{-6}秒	3600×10^{-6}秒
算法 6	n^4	0.16秒	2.56秒	-------秒
算法 7	2^n	1.05秒	12.73天	-------年
算法 8	$n!$	77147年	2.56×10^{34}年	2.64×10^{68}年

P & NP

P： **(Decision) problems solvable in Polynomial time**

NP： **Decision Problems verifiable in Polynomial time**

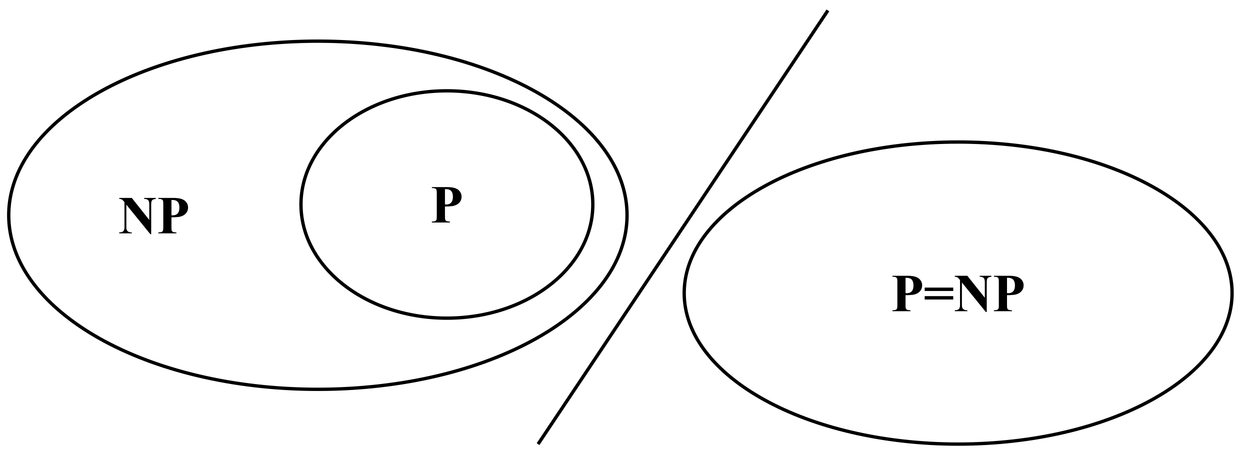
P & NP

P： (Decision) problems solvable in Polynomial time

NP： Decision Problems verifiable in Polynomial time

P & NP

P： **(Decision) problems solvable in Polynomial time**
NP： **Decision Problems verifiable in Polynomial time**

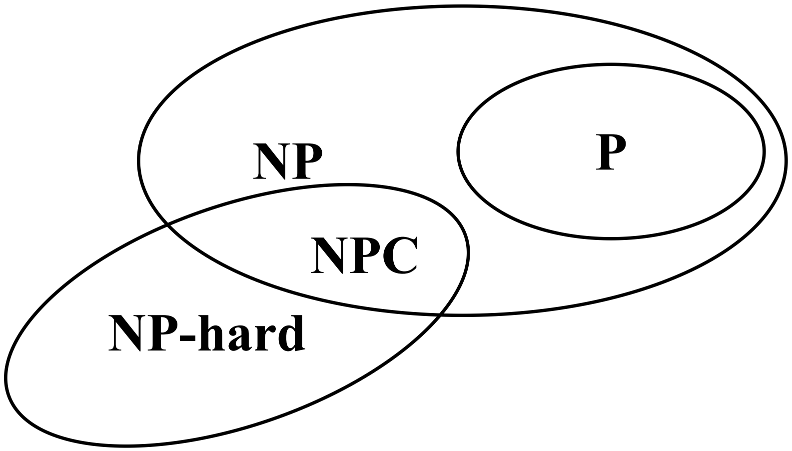


千禧年大奖难题
Millennium Prize Problems

P & NP

P： **(Decision) problems solvable in Polynomial time**

NP： **Decision Problems verifiable in Polynomial time**



Assume P≠NP

Minimal Spanning Trees and Metric TSP

Traveling salesman problem (TSP)
Given a complete graph with nonnegative edge costs, find a minimum cost cycle visiting every vertex exactly once.

Metric TSP
The edge costs satisfy triangle inequality.

The Metric TSP is an NP-hard problem.
If $\mathrm{P} \neq \mathrm{NP}$, we can't simultaneously have algorithms that
(1) find optimal solutions
(2) in polynomial-time
(3) for any instance.
At least one of these requirements must be relaxed in any approach to dealing with an NP-hard optimization problem.

Minimal Spanning Trees and Metric TSP

Traveling salesman problem (TSP)

Given a complete graph with nonnegative edge costs, find a minimum cost cycle visiting every vertex exactly once.

Metric TSP

The edge costs satisfy triangle inequality.

The Metric TSP is an NP-hard problem.

If $\mathrm{P} \neq \mathrm{NP}$, we can’t simultaneously have algorithms that

(1) find optimal solutions ⇒ 2-approximation algorithm

(2) in polynomial-time

(3) for any instance.

At least one of these requirements must be relaxed in any approach to dealing with an NP-hard optimization problem.

Minimal Spanning Trees and Metric TSP

Traveling salesman problem (TSP)

Given a complete graph with nonnegative edge costs, find a minimum cost cycle visiting every vertex exactly once.

Metric TSP

The edge costs satisfy triangle inequality.

Approximation Algorithms

Definition: An α-apppproximation algorithm for an optimization problem is a polynomial-time algorithm that for all instances of the problem produces a solution whose value is within a factor of α of the value of an optimal solution.

极小化问题

$$\sup_{\text{实例 } I} \frac{\text{近似算法解的值 Cost A (I)}}{\text{最优解的值 Opt (I)}} \leq \alpha$$

Minimal Spanning Trees and Metric TSP

A 2-approximation algorithm for the Metric TSP

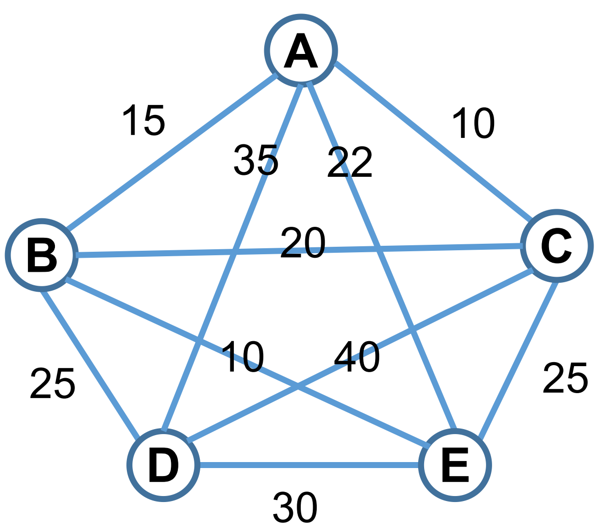
Input：A complete graph with nonnegative edge costs that satisfy the triangle inequality.
Ouptput: A cycle C visiting every vertex exactly once.

1. Find an MST, T of G.
2. Double every edge of the MST to obtain an Euler graph.
3. Find an Euler cycle, T_{eu} , on this graph.
4. Output the cycle that visits vertices of G in the order of their first appearance in T_{eu}. Let C be this cycle.

A 2-approximation algorithm for the Metric TSP

Input: A complete graph with nonnegative edge costs that satisfy the triangle inequality.
Ouptput: A cycle C visiting every vertex exactly once.

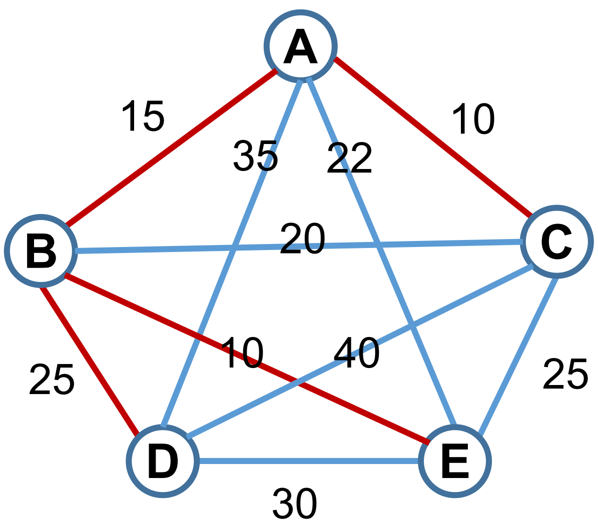
1. Find an MST, T of G.
2. Double every edge of the MST to obtain an Euler graph.
3. Find an Euler cycle, T_{eu} , on this graph.
4. Output the cycle that visits vertices of G in the order of their first appearance in T_{eu}. Let C be this cycle.



A 2-approximation algorithm for the Metric TSP

Input: A complete graph with nonnegative edge costs that satisfy the triangle inequality.
Ouptput: A cycle C visiting every vertex exactly once.

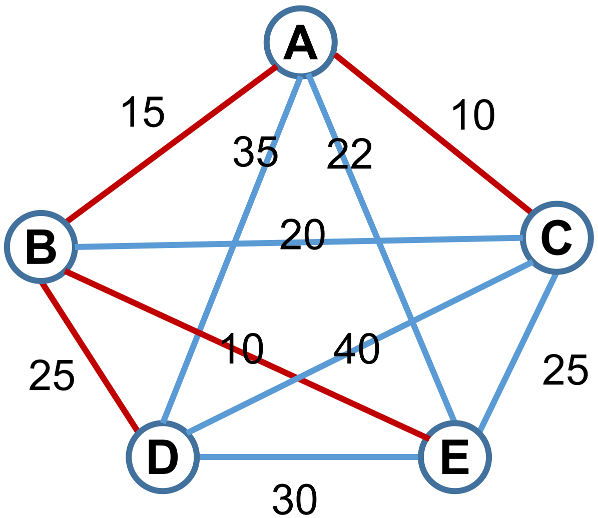
1. Find an MST, T of G.
2. Double every edge of the MST to obtain an Euler graph.
3. Find an Euler cycle, T_{eu} , on this graph.
4. Output the cycle that visits vertices of G in the order of their first appearance in T_{eu}. Let C be this cycle.

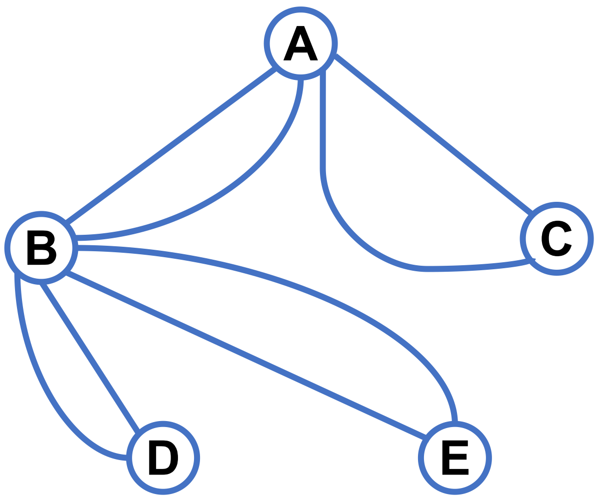


A 2-approximation algorithm for the Metric TSP

Input： A complete graph with nonnegative edge costs that satisfy the triangle inequality.
Ouptput: A cycle C visiting every vertex exactly once.

1. Find an MST, T of G.
2. Double every edge of the MST to obtain an Euler graph.
3. Find an Euler cycle, T_{eu} , on this graph.
4. Output the cycle that visits vertices of G in the order of their first appearance in T_{eu}. Let C be this cycle.



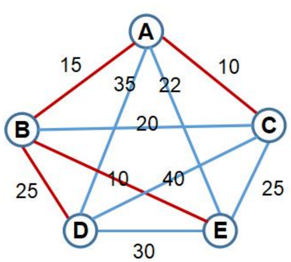


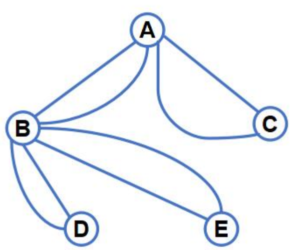
A 2-approximation algorithm for the Metric TSP

Input: A complete graph with nonnegative edge costs that satisfy the triangle inequality.
Ouptput: A cycle C visiting every vertex exactly once.

1. Find an MST, T of G.
2. Double every edge of the MST to obtain an Euler graph.
3. Find an Euler cycle, T_{eu} , on this graph.
4. Output the cycle that visits vertices of G in the order of their first appearance in T_{eu}. Let C be this cycle.

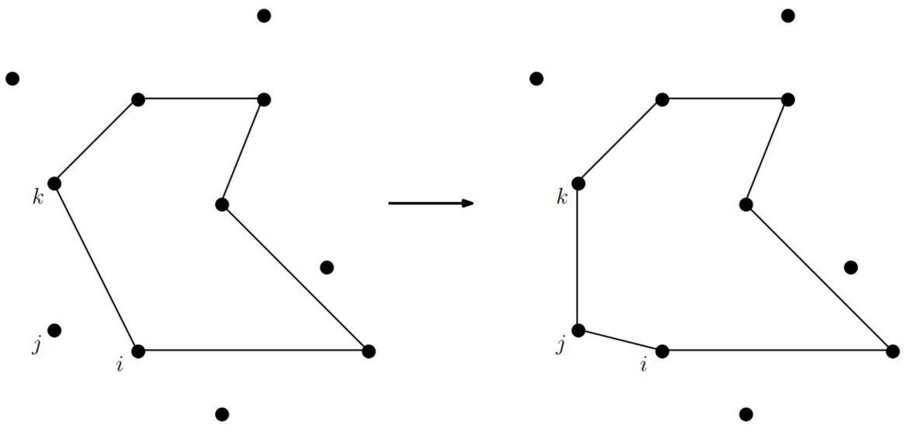
Proof





A 2-approximation algorithm for the Metric TSP

Here is a natural greedy heuristic to consider for the traveling salesman problem; this is often referred to as the *nearest addition algorithm*. Find the two closest cities, say i and j, and start by building a tour on that pair of cities; the tour consists of going from i to j and then back to i again. This is the first iteration. In each subsequent iteration, we extend the tour on the current subset S by including one additional city, until we include the full set of cities. In each iteration, we find a pair of cities $i \in S$ and $j \notin S$ for which the cost c_{ij} is minimum; let k be the city that follows i in the current tour on S. We add j to S, and insert j into the current tour between i and k. An illustration of this algorithm is shown in Figure 2.4.



A 2-approximation algorithm for the Metric TSP

Here is a natural greedy heuristic to consider for the traveling salesman problem; this is often referred to as the *nearest addition algorithm*. Find the two closest cities, say i and j, and start by building a tour on that pair of cities; the tour consists of going from i to j and then back to i again. This is the first iteration. In each subsequent iteration, we extend the tour on the current subset S by including one additional city, until we include the full set of cities. In each iteration, we find a pair of cities $i \in S$ and $j \notin S$ for which the cost c_{ij} is minimum; let k be the city that follows i in the current tour on S. We add j to S, and insert j into the current tour between i and k. An illustration of this algorithm is shown in Figure 2.4.