算法设计与分析 期末复习

20220109

基础知识

1.渐进记号

Θ记号

定义：

$$\Theta(g(n)) = \{f(n)|存在正常量c_1,c_2,n_0，使得对所有n \geq n_0，有0 \leq c_1g(n) \leq f(n) \leq c_2g(n)\}$$

表示渐进紧确界

O记号

定义：

$$O(g(n)) = \{f(n)|存在正常量c,n_0，使得对所有n \geq n_0，有0 \leq f(n) \leq cg(n)\}$$

表示渐进上界

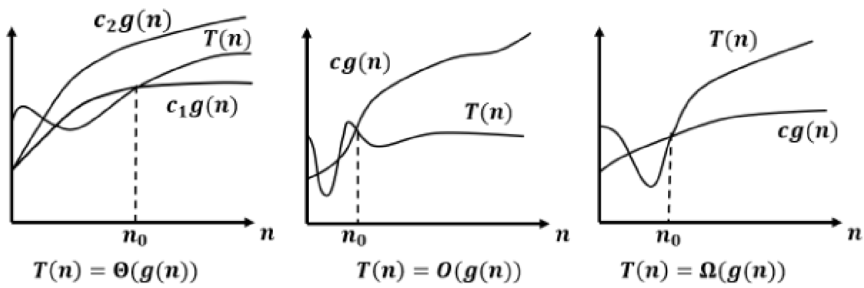
Ω记号

定义：

$$O(g(n)) = \{f(n)|存在正常量c,n_0，使得对所有n \geq n_0，有0 \leq cg(n) \leq f(n)\}$$

表示渐进下界

图例



2.三种情况分析

最好情况分析（Best Case）

- 对于输入n的最短可能运行时间

最差情况分析（Worst Case）

- 对于输入n的最长可能运行时间
- 通常使用的方法

平均（期望）情况分析（Average Case）

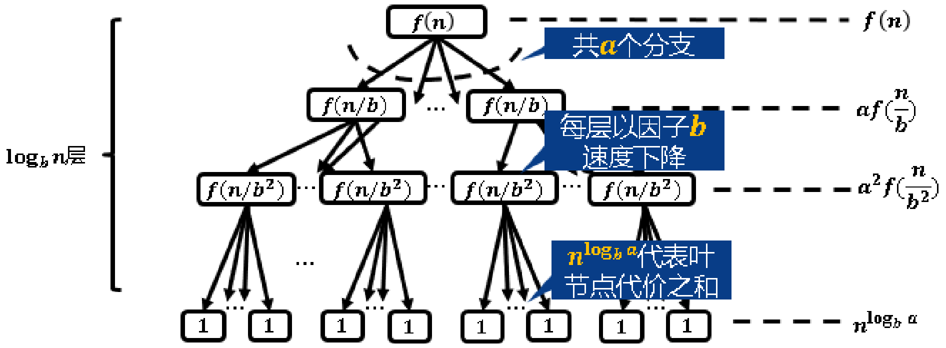
- 对于输入n的所有可能情况的平均运行时间

3. 递归式求解

主定理法

对于形如$T(n) = aT(\frac{n}{b}) + f(n)$的递归式，其解为：

$$T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n - 1} a^i f(\frac{n}{b^i})$$



进一步分析：

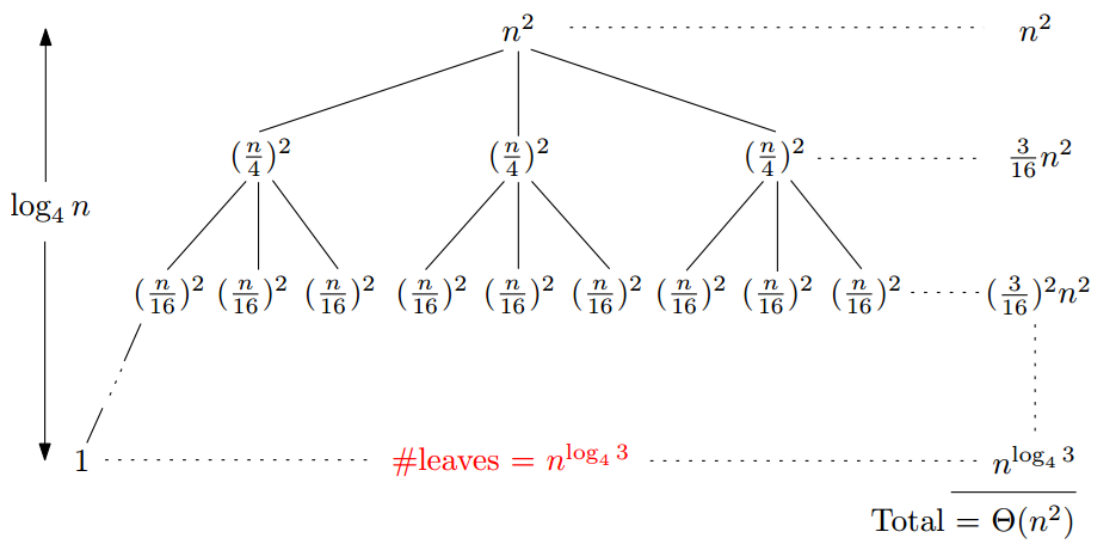
$$T(n) = \begin{cases} \Theta(f(n)) & , if\ f(n) = \Omega(n^{\log_b a + \epsilon}) \\ \Theta(n^{\log_b a} \log n) & , if\ f(n) = \Theta(n^{\log_b a}) \\ \Theta(n^{\log_b a}) & , if\ f(n) = O(n^{\log_b a - \epsilon}) \end{cases}$$

当$f(n)$形式为n^k时，可以简化主定理公式：

$$T(n) = aT(\frac{n}{b}) + n^k = \begin{cases} \Theta(n^k) & , if\ k > \log_b a \\ \Theta(n^k \log n) & , if\ k = \log_b a \\ \Theta(n^{\log_b a}) & , if\ k > \log_b a \end{cases}$$

递归树法

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$



和式求解

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

$$\sum_{i=1}^{n} i^2 = O(\log n^3)$$

$$\sum_{i=1}^{n} \frac{1}{i} = O(\log n)$$

分治算法

- 分解原问题
- 解决子问题
- 合并问题解

1. 归并排序

2. 最大子数组

3. 逆序计数

4. 多项式乘法

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{n-\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{n-\frac{n}{2}}$;
$Y(x) \leftarrow$PolyMulti2$(A_0(x) + A_1(x), B_0(x) + B_1(x))$;$//T(n/2)$
$U(x) \leftarrow$PolyMulti2$(A_0(x), B_0(x))$;$//T(n/2)$
$Z(x) \leftarrow$PolyMulti2$(A_1(x), B_1(x))$;$//T(n/2)$
return $(U(x) + [Y(x) - U(x) - Z(x)]x^{\frac{n}{2}} + Z(x)x^{2\frac{n}{2}})$;$//O(n)$

$$T(n) = O(n \log n)$$

5. 快速排序与划分

$$Quicksort(A, p, r)$$

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Sorted array $\boldsymbol{A}$
if $p < r$ **then**
    $q \leftarrow$Partition(A, p, r);
    Quicksort$(A, p, q-1)$;
    Quicksort$(A, q+1, r)$;
end
return A;

$$Partition(A, p, r)$$

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Index of the pivot after partition
$x \leftarrow A[r]$;//$A[r]$ is the pivot element
$i \leftarrow p-1$;
for $j \leftarrow p$ *to* $r-1$ **do**
    if $A[j] \leq x$ **then**
        $i \leftarrow i+1$;
        exchange $A[i]$ and $A[j]$;
    end
end
exchange$A[i+1]$ and $A[r]$;//Put pivot in position
return $i+1$;//$q \leftarrow i+1$

6. 随机选择

$Randomized-Partition(A,p,r)$

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: A random index in $[p..j]$
$j \leftarrow$random(p,r);
exchange $A[r]$ and $A[j]$;
Partition(A,p,r);
return j;

7. 基于比较的排序下界

在最坏情况下，任何基于比较的排序算法都需要做$\Omega(n\log n)$次比较

动态规划

- 问题结构分析
- 递推关系建立
- 自底向上计算
- 最优方案追踪

1. 0-1背包

Input: Allowed maximum weight $\boldsymbol{W}$, intermediate array from Knapsack $\boldsymbol{V}$
Output: Maximum value of any subset of items $\{1,2,...,n\}$ of weight at most W.
$K \leftarrow W$;
for $i \leftarrow n$ *to* 1 **do**
    if *keep*$[i,K]$ *is equal to* 1 **then**
        Output i;
        $K \leftarrow K - w[i]$;
    end
end

Let $V[0..n, 0..W]$ and $keep[0..n, 0..W]$ be two new 2-dimension arrays;
for $i \leftarrow 0$ *to* W **do**
    $V[0, i] \leftarrow 0$;
end
for $i \leftarrow 1$ *to* n **do**
    for $j \leftarrow 0$ *to* W **do**
        if $(w[i] \leq w)$ *and* $(v[i] + V[i-1, w - w[i]] > V[i-1, j])$ **then**
            $V[i, j] \leftarrow \max\{V[i-1, j], v[i] + V[i-1, w - w[i]]\}$;
            $keep[i, j] \leftarrow 1$;
        end
        else
            $V[i, j] \leftarrow V[i-1, j]$;
            $keep[i, j] \leftarrow 0$;
        end
    end
end
$K \leftarrow W$;
for $i \leftarrow n$ *to* 1 **do**
    if $keep[i, K]$ *is equal to* 1 **then**
        Output i;
        $K \leftarrow K - w[i]$;
    end
end
return $V[n, W]$;

$T(n) = O(nW)$

2. 最大子数组

3. 最长公共子序列

$Longest - Commom - Subsequence(X, Y)$

Input: Two strings $\boldsymbol{X}$,$\boldsymbol{Y}$.
Output: Longest common subsequence of X and Y.
$m \leftarrow$length(X);
$n \leftarrow$length(Y);
Let $d[0..m, 0..n]$ and $p[0..m, 0..n]$ be two new 2-dimension arrays;
//Initializaiton
for $i \leftarrow 0$ *to* m **do**
    $d[i, 0] \leftarrow 0$;
end
for $j \leftarrow 0$ *to* n **do**
    $d[0, j] \leftarrow 0$;
end

```
//Dynamic Programming
for i ← 1 to m do
    for j ← 1 to n do
        if x_i is equal to y_j then
            d[i, j] ← d[i − 1, j − 1] + 1;
            p[i, j] ←"LU";//"LU" indicates left up arrow.
        end
        else if d[i − 1, j] ≥ d[i, j − 1] then
            d[i, j] ← d[i − 1, j];
            p[i, j] ←"U";//"U" indicates up arrow.
        end
        else
            d[i, j] ← d[i, j − 1];
            p[i, j] ←"L";//"L" indicates left arrow.
        end
    end
end
return d, p;
```

$Print - LCS(p, X, i, j)$

```
Input: Array p generated from Longest-Common-Subsequence, string
       X, index i and j.
Output: Output the longest common subsequence of X[1..i] and Y[1..j].
if i is equal to 0 or j is equal to 0 then
    return NULL;
end
if p[i, j] is equal to "LU" then
    Print-LCS(p, X, i − 1, j − 1);
    print x_i;
end
else if p[i, j] is equal to "U" then
    Print-LCS(p, X, i − 1, j);
end
else
    Print-LCS(p, X, i, j − 1);
end
```

$T(n) = O(mn)$

4.最长公共子串

$Longest - Commom - Substring(X, Y)$

Input: Two strings $\boldsymbol{X}$,$\boldsymbol{Y}$.
Output: Longest common substring of X and Y.
$m \leftarrow$length(X);
$n \leftarrow$length(Y);
Let $d[0..m, 0..n]$ be a new 2-dimension array;
$l_{max} \leftarrow 0$;
$p_{max} \leftarrow 0$;
//Initializaiton
for $i \leftarrow 0$ *to* m **do**
  $d[i,0] \leftarrow 0$;
end
for $j \leftarrow 0$ *to* n **do**
  $d[0,j] \leftarrow 0$;
end

//Dynamic Programming
for $i \leftarrow 1$ *to* m **do**
  for $j \leftarrow 1$ *to* n **do**
    if $x_i \neq y_j$ **then**
      $d[i,j] \leftarrow 0$;
    end
    else
      $d[i,j] \leftarrow d[i-1, j-1] + 1$;
      if $d[i,j] > l_{max}$ **then**
        $l_{max} \leftarrow d[i,j]$;
        $p_{max} \leftarrow i$;
      end
    end
  end
end
return l_{max}, p_{max};

$Print - LCSubstring(X, l_{\max}, p_{\max})$

Input: String $\boldsymbol{X}$, $\boldsymbol{l_{max}}$ and $\boldsymbol{p_{max}}$ are generated from Longest-Common-Substring.
Output: Output the longest common substring of $X[1..i]$ and $Y[1..j]$.
if l_{max} *is equal to 0* **then**
  return *NULL*;
end
for $i \leftarrow (p_{max} - l_{max} + 1)$ *to* p_{max} **do**
  print x_i;
end

$T(n) = O(mn)$

5.最小编辑距离

$Minimum - Edit - Distance(X, Y)$

Input: Two strings $\boldsymbol{X}$,$\boldsymbol{Y}$.
Output: Minimum edit distance of X and Y.
$m \leftarrow$length(X);
$n \leftarrow$length(Y);
Let $d[0..m, 0..n]$ and $p[0..m, 0..n]$ be two new 2-dimension arrays;
//Initializaiton
for $i \leftarrow 0$ *to* m **do**
  $d[i, 0] \leftarrow i$;
  $p[i, 0] \leftarrow$"U";
end
for $j \leftarrow 0$ *to* n **do**
  $d[0, j] \leftarrow j$;
  $p[0, j] \leftarrow$"L";
end

//Dynamic Programming
for $i \leftarrow 1$ *to* m **do**
  for $j \leftarrow 1$ *to* n **do**
    if x_i *is not equal to* y_j **then**
      $c \leftarrow 1$;
    end
    else
      $c \leftarrow 0$;
    end
    if $d[i-1, j-1] + c \leq d[i-1][j] + 1$ *and*
     $d[i-1, j-1] + c \leq d[i][j-1] + 1$ **then**
      $d[i, j] \leftarrow d[i-1, j-1] + c$;
      $p[i, j] \leftarrow$"LU";//"LU" indicates left up arrow.
    end
    else if $d[i, j-1] + 1 < d[i-1][j] + 1$ *and*
     $d[i, j-1] + 1 < d[i-1, j-1] + c$ **then**
      $d[i, j] \leftarrow d[i, j-1] + 1$;
      $p[i, j] \leftarrow$"L";//"L" indicates up arrow.
    end
    else
      $d[i, j] \leftarrow d[i-1, j] + 1$;
      $p[i, j] \leftarrow$"U";//"U" indicates left arrow.
    end
  end
end
return d, p;

$$Print - MED(p, X, i, j)$$

```
Input: Array p generated from Minimum-Edit-Distance, string X, index
       i and j.
Output: Output the sequence of opeations.
if i is equal to 0 and j is equal to 0 then
 |  return NULL;
end
if p[i, j] is equal to "LU" then
 |  Print-MED(p, X, i − 1, j − 1);
 |  if x_i is equal to y_j then
 |   |  print "Do nothing"
 |  end
 |  else
 |   |  print "Substitue x_i with y_j;
 |  end
end
else if p[i, j] is equal to "U" then
 |  Print-MED(p, X, i − 1, j);
 |  print "Delete x_i";
end
else
 |  Print-MED(p, X, i, j − 1);
 |  print "Insert y_j"
end
```

$T(n) = O(mn)$

6. 钢条切割

7. 矩阵链乘法

```
Let m[1..n, 1..n] and s[1..n, 1..n] be two 2-dimension arrays;
for i ← 1 to n do
 |  m[i, i] ← 0;
end
for l ← 2 to n do
 |  for i ← 1 to n − l + 1 do
 |   |  j ← i + l − 1;
 |   |  m[i, j] ← ∞;
 |   |  for k ← i to j − 1 do
 |   |   |  q ← m[i, k] + m[k + 1, j] + p[i − 1] ∗ p[k] ∗ p[j];
 |   |   |  if q < m[i, j] then
 |   |   |   |  m[i, j] ← q;
 |   |   |   |  s[i, j] ← k;
 |   |   |  end
 |   |  end
 |  end
end
return m[1, n] and s;
```

$T(n) = O(n^3)$

贪心算法

- 提出贪心策略
- 证明策略正确

  一个常见的方法是：假设最优方案为X，而贪心算法得到的结果为Y。若X与Y不同，则能够在保证X不变差的情况下将X转化为Y。

1. 部分背包

Input: Value array $\boldsymbol{v}$ and weight array $\boldsymbol{w}$ of $\boldsymbol{n}$ items, capacity of knapsack $\boldsymbol{K}$.
Output: Solution of maximum value.

Let r[1..n],x[1..n] be two new arrays;
for $i \leftarrow 1$ *to* n **do**
  $r[i] \leftarrow v[i]/w[i]$;
  $x[i] \leftarrow 0$;
end
Sort the items in decreasing order of their ratios r, rename the items if necessary so that the sorted order of items is $\langle 1, 2, ..., n \rangle$;
$j \leftarrow 0$;
while $K > 0$ *and* $j \leq n$ **do**
  $j \leftarrow j + 1$;
  if $K > w[j]$ **then**
    $x[j] \leftarrow 1$;
    $K \leftarrow K - w[j]$;
  end
  else
    $x[j] \leftarrow k/w[j]$;
    break;
  end
end
return x;

2. 霍夫曼编码

Input: An alphabet $\boldsymbol{A}$ with frequency distribution.
Output: Huffman tree.

$n \leftarrow |A|$;
$Q \leftarrow$ a new Priority Queue of A;
for $i \leftarrow 1$ *to* $n - 1$ **do**
  // Why $n - 1$?
  $z \leftarrow$ a new node;
  $z.left \leftarrow$Extract-Min(Q);
  $z.right \leftarrow$Extract-Min(Q);
  $z.freq \leftarrow z.left.freq + z.right.freq$;
  Insert(Q, z);
end
return *Extract-Min(Q)*;

$$T(n) = O(n \log n)$$

3. 活动选择

Input: a set of activities $A = a_1, a_2 \ldots, a_n$
Output: the largest subset of A that do not overlap
Sort activities in increasing order of finishing time;
$P = a_1$; `// insert the activity with earliest finishing time`
$k = 1$; `// index to the last activity in A`
for $i \leftarrow 2$ ***to*** n **do**
  if $s[i] \geq f[k]$ **then**
    `//` i `starts after` k `finishes - no overlap`
    $P \leftarrow P \cup a_i$;
    $k \leftarrow i$;
  end
end
return P;

时间复杂度主要来自排序算法

$T(n) = O(n \log n)$

带权重的活动选择——动态规划

Input: a set of activities $A = a_1, a_2, \cdots, a_n$
Output: the max weight of any subset of mutually compatible activities
Sort activities by finishing time and renumber so that $f_1 \leq f_2 \leq \cdots \leq f_n$;
Compute $p[1], p[2], \cdots, p[n]$ via binary search;
$OPT[0] \leftarrow 0$;
for $j = 1$ ***to*** n **do**
  $OPT[j] \leftarrow \max\{OPT[j-1], w_j + OPT[p[j]]\}$;
end
return $OPT[n]$;

$T(n) = O(n \log n)$

4. 最小生成树

5. 单源最短路径

图算法

1. 图的基本概念

2. 广度优先搜索

- color[u]: the color of each vertex visited

> *WHITE: undiscovered*
>
> *GRAY: discovered but not finished processing*
>
> *BLACK: finished processing*

- pred[u]: the predecessor pointer

> *pointing back to the vertex from which u was discovered*

- d[u]: the distance from the source to vertex u

$BFS(G)$

```
Input: A graph G
Output: None
//Initialize
for u in V do
    color[u] ←WHITE;//undiscovered
    pred[u] ←NULL;//no predecessor
end
for u in V do
    //start a new tree
    if color[u] is equal to WHITE then
        BFSVisit(u);
    end
end
```

$BFSVisit(s)$

```
Input: A vertex s
Output: None
color[s] ← GRAY,d[s] ← 0;
Q ← ∅,Enqueue(Q, s);
while Q ≠ ∅ do
    u ←Dequeue(Q);
    for v ∈ Adj[u] do
        if color[v] ←WHITE then
            color[v] ←GRAY;
            d[v] ← d[u] + 1;
            pred[v] ← u;
            Enqueue(Q, v);
        end
    end
    color[u] ←BLACK;
end
```

$T(n) = O(V + E)$

3.深度优先搜索

- color[u]: the color of each vertex visited

> *WHITE: undiscovered*
>
> *GRAY: discovered but not finished processing*
>
> *BLACK: finished processing*

- pred[u]: the predecessor pointer

> *pointing back to the vertex from which u was discovered*

- d[u]: the discovery time

> *a counter indicating when vertex u is discovered*

- f[u]: the finishing time

> *a counter indicating when the processing of vertex u (and all its descendants) is finished*

$DFS(G)$

```
Input: A graph G
Output: None
for u in V do
    color[u] ←WHITE;//undiscovered
    pred[u] ←NULL;//no predecessor
end
time ← 0;
for u in V do
    //start a new tree
    if color[u] is equal to WHITE then
        DFSVisit(u);
    end
end
```

$DFSVisit(u)$

```
Input: A vertex u
Output: None
color[u] ← GRAY;//u is discovered
d[u] ← ++time;//u's discovery time
for v ∈ Adj(u) do
    //Visit undiscovered vertex
    if color[v] is equal to WHITE then
        pred[v] ← u;
        DFSVisit(v);
    end
end
color[u] ←BLACK;//u has finished
f[u] ← ++time;//u's finish time
```

$T(n) = O(V + E)$

4. 环路检测

5. 拓扑排序

```
Input: A graph G
Output: None
Initialize Q to be an empty queue;
for u ∈ V do
    if u.in_degree is equal to 0 then
        //Find all starting vertices
        Enqueue(Q, u);
    end
end
while Q is not empty do
    u ←Dequeue(Q);
    Output u;
    for v ∈ Adj(u) do
        //remove u's outgoing edges
        v.in_degree ← v.in_degree − 1;
        if v.in_degree is equal to 0 then
            Enqueue(Q, v);
        end
    end
end
```

出队顺序即为输出顺序

$$T(n) = O(V + E)$$

6.强联通分量

```
Input: A directed graph G
Output: The set of strongly connected components R
R ← {};// set of SCCs
G^R ←reverse graph of G;
L^R ←DFS-b(G^R);// Perform DFS
L ←reverse order of L^R;
for u ∈ L do
    if color[u] is equal to WHITE then
        Lscc ←DFSVisit(G,u);// Perform DFS starting at u
        R ← R∪Set(Lscc);
    end
end
return R;
```

$$T(n) = O(V + E)$$

连通图是有向无环图

7.最小生成树

- Prim算法

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$
Output: None
Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;
for $u \in V$ **do**
  $color[u] \leftarrow$ WHITE,$key[u] \leftarrow +\infty$;// Initialize
end
$key[r] \leftarrow 0$, $pred[r] \leftarrow$ NULL;// Start at root vertex
$Q \leftarrow$ new PriQueue(V);// put vertices in Q
while Q *is nonempty* **do**
  $u \leftarrow Q$.Extract-Min();// lightest edge
  for $v \in adj[u]$ **do**
    if $(color[v] \leftarrow WHITE)\&\&(w[u,v] < key[v])$ **then**
      $key[v] \leftarrow w[u,v]$;// new lightest edge
      Q.Decrease-Key$(v, key[v])$;
      $pred[v] \leftarrow u$;
    end
  end
  $color[u] \leftarrow$ BLACK;
end

结果由$pred[\ ]$表示

$key[\ ]$之和为选中的边权重之和

$T(n) = O(E \log V)$

- Kruskal算法

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G
Output: MST of G
Sort E in increasing order by weight w;// $O(|E| \log |E|)$
// After sorting $E = \langle \{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_{|E|}, v_{|E|}\} \rangle$
$A \leftarrow \{\}$;
for $u \in V$ **do**
  Create-Set(u);// $O(|V|)$
end
for $e_i \in E$ **do**
  // $O(|E| \log |V|)$
  if *Find-Set(u_i)≠Find-Set(v_i)* **then**
    add $\{u_i, v_i\}$ to A;
    Union(u_i, v_i);
  end
end
return A;

$T(n) = O(E \log V)$

8.单源最短路径

- BFS

  适用于无权图

- Dijkstra算法

  要求所有边权重为非负值

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$
Output: None
for $u \in V$ **do**
  $d[u] \leftarrow \infty, color[u] \leftarrow$ WHITE;// Initialize
end
$d[s] \leftarrow 0$;
$pred[s] \leftarrow$ NULL;
$Q \leftarrow$queue with all vertices;
while *Non-Empty(Q)* **do**
  // Process all vertices
  $u \leftarrow$Extract-Min(Q);// Find new vertex
  for $v \in Adj[u]$ **do**
    if $d[u] + w(u,v) < d[v]$ **then**
      // If estimate improves
      $d[v] \leftarrow d[u] + w(u,v)$;// relax
      Decrease-Key$(Q, v, d[v])$;
      $pred[v] \leftarrow u$;
    end
  end
  $color[u] \leftarrow$BLACK;
end

$$T(n) = O(E \log V)$$

- Bellman-Ford算法

$$Relax(u, v)$$

Input: Update estimation of $\boldsymbol{u}$ according to distance of $\boldsymbol{v}$
Output: None
if $d[u] + w(u,v) < d[v]$ **then**
  $d[v] \leftarrow d[u] + w(u,v)$;
  $pred[v] \leftarrow u$;
end

$$Bellman - Ford(G, w, s)$$

Input: A directed graph $\boldsymbol{G}$,weights $\boldsymbol{w}$, and the source vertex $\boldsymbol{s}$
Output: Return FALSE if G contains negative cycle, return TRUE if shortest paths from s to any other vertices obtained.
for $u \in V$ **do**
  $d[u] \leftarrow \infty, pred[u] \leftarrow$NIL;// Initialize
end
for $i \leftarrow 1$ *to* $|V| - 1$ **do**
  for $e \in E$ **do**
    RELAX(u, v, w);
  end
end
for $e \in E$ **do**
  if $d[v] > d[u] + w(u,v)$ **then**
    return *FALSE*;
  end
end
return *TRUE*;

$$T(n) = O(EV)$$

处理难问题

1. 问题分类

- P

  在多项式时间内可以解决的问题，即可以在$O(n^k)$内解决

- NP

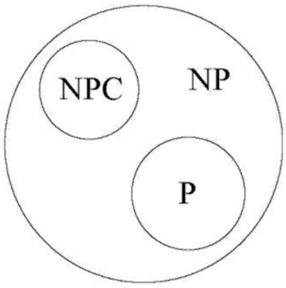
在多项式时间内可以被证明的问题

- NPC

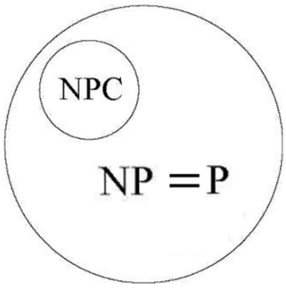
  NP完全问题

说明:

- $P \subset NP$
- 如果$NP \neq P$



- 如果$NP = P$



2. 证明问题为NPC问题

- 证明Y∈NP
- 寻找一个已知的NPC问题X，并证明X≤p Y.

> 如何证明***X≤p Y?***
>
> 使用*Y*的结果解决*X*
>
> - 对问题*X*的任意一个输入*x*, 将其映射为*Y*的一个输入*f*(*x*)。
> - 证明问题*X*在输入*x*的条件下返回*"yes"*当且仅当问题*Y*在输入*f*(*x*)的条件下返回*"yes"*.

3. NPC问题举例

- SAT问题（布尔可满足性问题）

  某一个布尔表达式是不是“可满足”的问题

  “可满足”的意思是存在一组“真值赋值”使得布尔表达式为真

  3-SAT问题

  某个具有特殊形式的布尔表达式是否可满足的问题

  特殊形式指"3合取范式"或"3-CNF”

- DCLIQUE（团问题）

  寻找图中规模最大的团

  团：完全子图

- Decision Vertex Cover (DVC)（顶点覆盖问题）

寻找图的最小顶点覆盖

顶点覆盖V：原图的每一条都有至少一个端点在V内

- Decision Independent Set (DIS)（独立集问题）

  寻找图的最大独立集

  独立集V：其内任意一对顶点之间都没有边相连