整理思路：

2016-11-17

1、又看到了 NP 问题，找了一些资料，然后整理理解一下，便于以后的翻阅。

2、先看定义：直接上图，看不懂下面的图，真的看不懂。

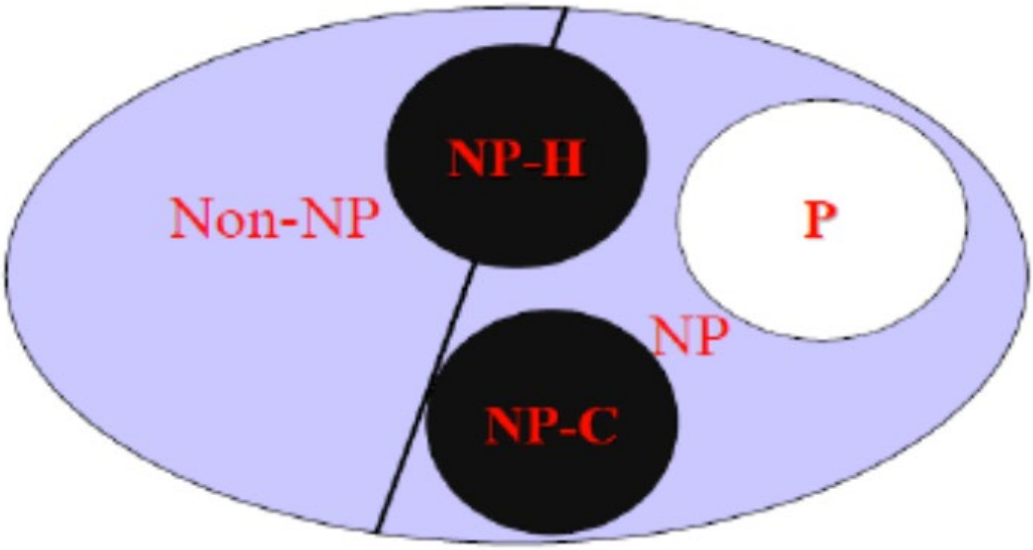


图 1 各类问题的集合论关系

说实话，上面的这个图好像有点问题。NP-Hard 是否包含 NPC？

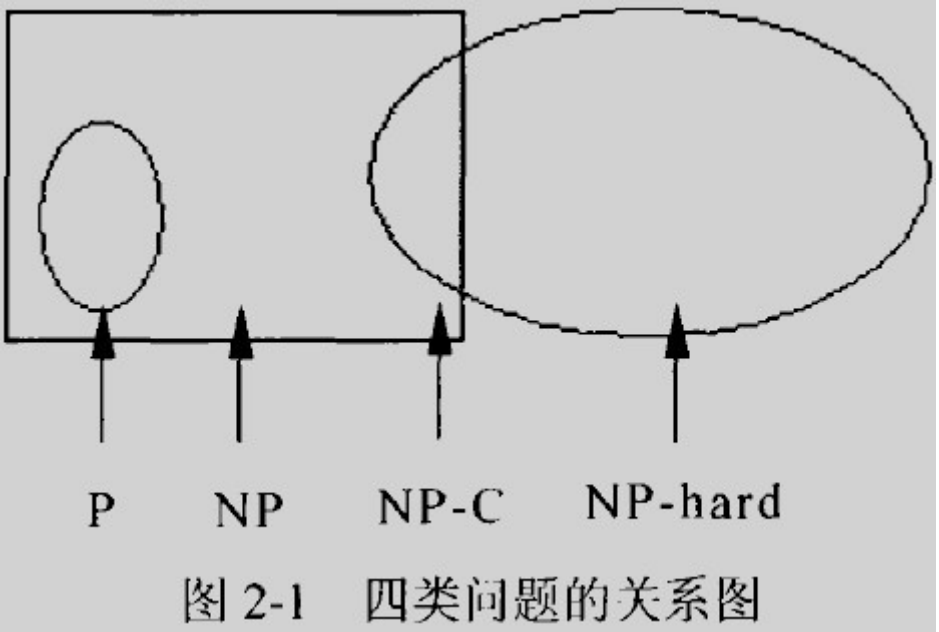
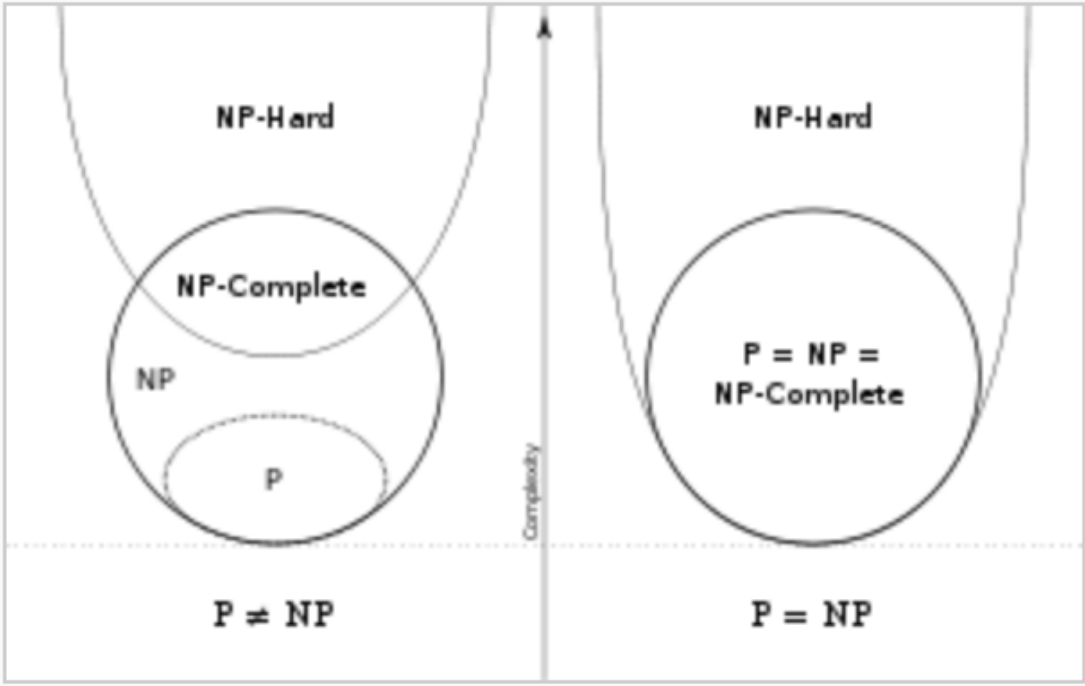


图 2-1　四类问题的关系图



Euler diagram for P, NP, NP-complete, and NP-hard set of problems. The existence of problems within **NP** but outside both **P** and **NP**-complete, under the P≠NP assumption, was established by Ladner.[1]

3、上图直接把四类问题给出来了，但是真的不理解。因此要明

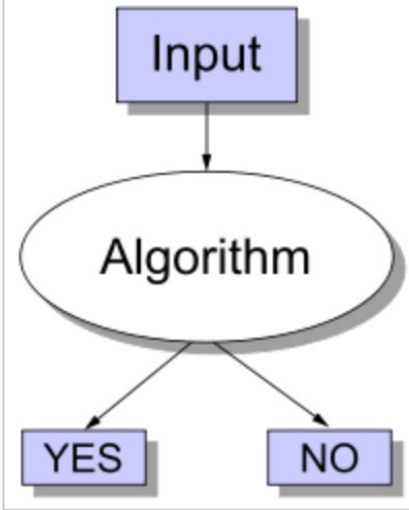
白，问题是怎么提出来的，或者说怎么引申出来的。看例子，截图：

在一个周六的晚上，你参加了一个盛大的晚会。由于感到局促不安，你想知道这一大厅中是否有你已经认识的人。你的主人向你提议说，你一定认识那位正在甜点盘附近角落的女士罗丝。不费一秒钟，你就能向那里扫视，并且发现你的主人是正确的。然而，如果没有这样的暗示，你就必须环顾整个大厅，一个个地审视每一个人，看是否有你认识的人。

这里牵涉出一个概念，就是这个问题是让你回答：有还是没有，或者对还是不对（“针对这里面有没有你认识的人”）。这个问题，就是“判定问题”。看截图：

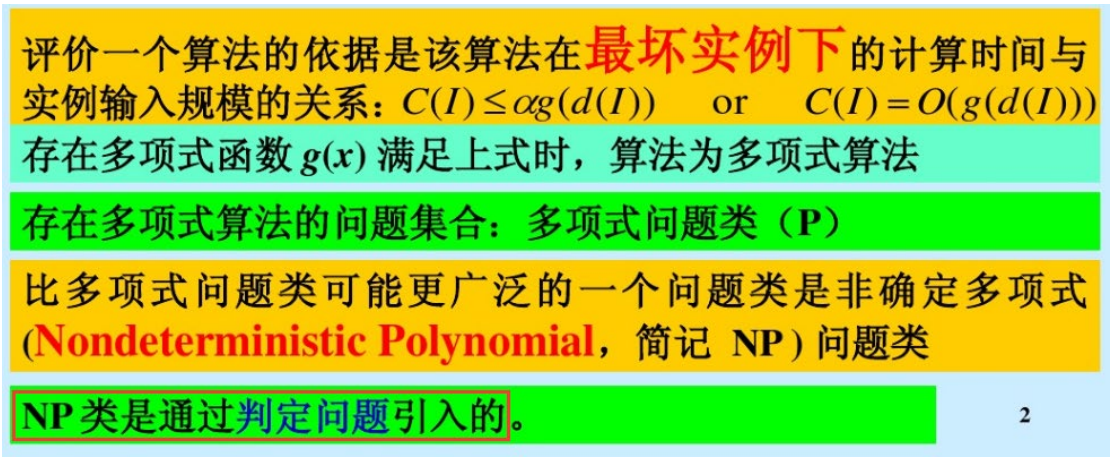
定义2.1 如果一个问题的每一个实例只有“是”或“否”两种答案，则称这个问题为判定问题(Decision / recognition / feasibility problem). 称有肯定答案的实例为“是”实例(yes-instance). 称答案为“否”的实例为“否”实例或非“是”实例(no-instance).

继续看英文的截图：



A *decision problem* has only two possible outputs, *yes* or *no* (or alternately 1 or 0) on any input.

4、那么 NP 和这个“判定问题”有毛关系呢？看截图说明：



NP 问题是通过判定问题引入的。怎么引入的，我们分析一下。分析不来啊，没有办法，从另外一个角度去引入吧。

5、就从字面意义上去理解，确定和非确定。看截图：

什么是NP(非确定性问题)呢？有些计算问题是确定性的，比如加减乘除之类，你只要按照公式推导，按部就班一步步来，就可以得到结果。但是，有些问题是无法按部就班直接地计算出来。比如，找大质数的问题。有没有一个公式，你一套公式，就可以一步步推算出来，下一个质数应该是多少呢？这样的公式是没有的。再比如，大的合数分解质因数的问题，有没有一个公式，把合数代进去，就直接可以算出，它的因子各自是多少？也没有这样的公式。

简单说，有些问题，你可以通过确定的公式和某种算法，得到结果。相反的，有些问题，你是没有确定的方法得到答案，你只有通过枚举法，但是这样的话，可能会无穷的，因此，计算量是非常大的。那只能猜测了，运气好的话，一次就中，运气不好的话，...。因此这里就有了两种说法：一个 P 类问题，是我可以在“多项式时间”里面，通过每种方法得到解；一个 NP 类问题，是我可以在“多项式时间”里面，去验证我给出来的一个解，是不是我要的解，或者说是对的还是错的（回到了“判定问题”这个概念）。那就要举例说明，先看 NP 问题，截图上：

NP问题是指可以在多项式的时间里验证一个解的问题。NP问题的另一个定义是，可以在多项式的时间里猜出一个解的问题。比方说，我RP很好（^.^），在程序中需要枚举时，我可以一猜一个准。现在某人拿到了一个求最短路径的问题，问从起点到终点是否有一条小于100个单位长度的路线。它根据数据画好了图，但怎么也算不出来，于是来问我：你看怎么选条路走得最少？我说，我RP很好，肯定能随便给你指条很短的路出来。然后我就胡乱画了几条线，说就这条吧。那人按我指的这条把权值加起来一看，嘿，神了，路径长度98，比100小。于是答案出来了，存在比100小的路径。别人会问他这题怎么做出来的，他就可以说，因为我找到了一个比100小的解。在这个题中，找一个解很困难，但验证一个解很容易。验证一个解只需要O(n)的时间复杂度，也就是说我可以花O(n)的时间把我猜的路径的长度加出来。那么，只要我RP好，猜得准，我一定能在多项式的时间里解决这个问题。我猜到的方案总是最优的，不满足题意的方案也不会来骗我去选它。这就是NP问题。

6、继续上面的问题，根据上面的例子，我们可以得知，我随意给出一个解，然后根据 n 次相加，就可以得到总的长度，然后我就可以知道这个解是不是小于 100。也就是，我可以去验证这个解是不是。那么回到 NP 问题的解释说明：“可以在多项式的时间里验证一个解的问题”。这里牵涉出一个概念：多项式的时间。不好理解。可以从编程的角度去理解，在这里的例子相当于循环了 n 次。比如，有 20 条边，那么 n 就等于 20，然后将 20 条边进行相加。具体点，如果把 20 条边放进一个 array 里面 E 的话，那就是有 E（0）到 E（20），然后一个 for 循环，就可以得到最后的值了，然后看看是不是小于 100。讲了这么多，这和多项式的时间有什么关系啊？因为不好理解，所以就用代码的角度去说明，这个就有点像多重循环，多重循环就相当于乘法。

7、继续多项式的时间。一个 P 类问题是可以用多项式时间求解的。上截图：

Some examples of polynomial time algorithms:

- The selection sort sorting algorithm on n integers performs An^2 operations for some constant A. Thus it runs in time $O(n^2)$ and is a polynomial time algorithm.
- All the basic arithmetic operations (addition, subtraction, multiplication, division, and comparison) can be done in polynomial time.
- Maximum matchings in graphs can be found in polynomial time.

看排序的这个例子：

Example [edit]

Here is an example of this sort algorithm sorting five elements:

```
64 25 12 22 11 // this is the initial, starting state of the array

11 25 12 22 64 // sorted sublist = {11}

11 12 25 22 64 // sorted sublist = {11, 12}

11 12 22 25 64 // sorted sublist = {11, 12, 22}

11 12 22 25 64 // sorted sublist = {11, 12, 22, 25}

11 12 22 25 64 // sorted sublist = {11, 12, 22, 25, 64}
```

代码如下：

Implementation [edit]

```
/* a[0] to a[n-1] is the array to sort */
int i,j;
int n;

/* advance the position through the entire array */
/*   (could do j < n-1 because single element is also min element) */
for (j = 0; j < n-1; j++)
  {
    /* find the min element in the unsorted a[j .. n-1] */

    /* assume the min is the first element */
    int iMin = j;
    /* test against elements after j to find the smallest */
    for ( i = j+1; i < n; i++) {
        /* if this element is less, then it is the new minimum */
        if (a[i] < a[iMin]) {
            /* found new minimum; remember its index */
            iMin = i;
        }
    }

    if(iMin != j)
    {
        swap(a[j], a[iMin]);
    }
}
```

这个时候，我们可以看到，是二重循环，这个是多项式的时间。极端一下，n 有 300 个，那么极端情况下，就是 300*300 次，有什么多的，数学表达式为：看截图

It has $O(n^2)$ time complexity

因此，排序问题是 P 类问题。它的时间复杂度属于“多项式级别的复杂度”。那么就应该有“非多项式级别的复杂度”。

8、时间复杂度分成两种：上截图

还是先用几句话简单说明一下时间复杂度。时间复杂度并不是表示一个程序解决问题需要花多少时间，而是当问题规模扩大后，程序需要的时间长度增长得有多快。也就是说，对于高速处理数据的计算机来说，处理某一个特定数据的效率不能衡量一个程序的好坏，而应该看当这个数据的规模变大到数百倍后，程序运行时间是否还是一样，或者也跟着慢了数百倍，或者变慢了数万倍。不管数据有多大，程序处理花的时间始终是那么多的，我们就说这个程序很好，具有O(1)的时间复杂度，也称常数级复杂度；数据规模变得有多大，花的时间也跟着变得有多长，这个程序的时间复杂度就是O(n)。比如：找n个数中的最大值；而像冒泡排序插入排序等。数据扩大2倍，时间变慢4倍的，属于O(n^2)的复杂度。还有一些穷举类的算法，所需时间长度成几何阶数上涨，这就是O(a^n)的指数级复杂度，甚至O(n!)的阶乘级复杂度。不会存在O(2*n^2)的复杂度，因为前面的那个“2”是系数，根本不会影响到整个程序的时间增长。同样地，O(n^3+n^2)的复杂度也就是O(n^3)的复杂度。因此，我们会说，一个O(0.01*n^3)的程序的效率比O(100*n^2)的效率低，尽管在n很小的时候，前者优于后者，但后者时间随数据规模增长得慢，最终O(n^3)的复杂度将远远超过O(n^2)。我们也说，O(n^100)的复杂度小于O(1.01^n)的复杂度。

从上面的截图，可以继续分类：截图上

杂度。

容易看出，前面的几类复杂度被分为两种级别，其中后者的复杂度无论如何都远远大于前者：一种是O(1),O(log(n)),O(n^a)等，我们把它叫做多项式级的复杂度，因为它的规模n出现在底数的位置；另一种是O(a^n)和O(n!)型复杂度，它是非多项式级的。其复杂度计算机往往不能承受。当我们在解决一个问题时，我们选择的算法通常都需要是多项式级的复杂度。非多项式级的复杂度需要的时间太多，往往会超时，除非是数据规模非常小。

因此，解决问题的时候，需要的是多项式级别的算法，所以，我们可以看到，其实算法大多是循环的，那个就是多项式级别的。*那么我们不还是在用一些算法求解 NP 问题吗？这个问题的解答应该是这样的，我们是用一些搜索算法来近似的求解 NPC 问题而已。（这个以后再说）*

9、继续 P 类问题和 NP 类问题。我们知道了他们之间的区别，那么怎么区分他们呢？NP 类是找不到一个有效的算法求解，但是可以验证（多项式的时间）是否正确；P 类是有一种有效的算法求解（多项式的时间，计算机求解的时候大部分都是多项式时间

吧），它肯定也可以去验证的啊，因此 P 类应该属于 NP 类的吧。看原来的图：

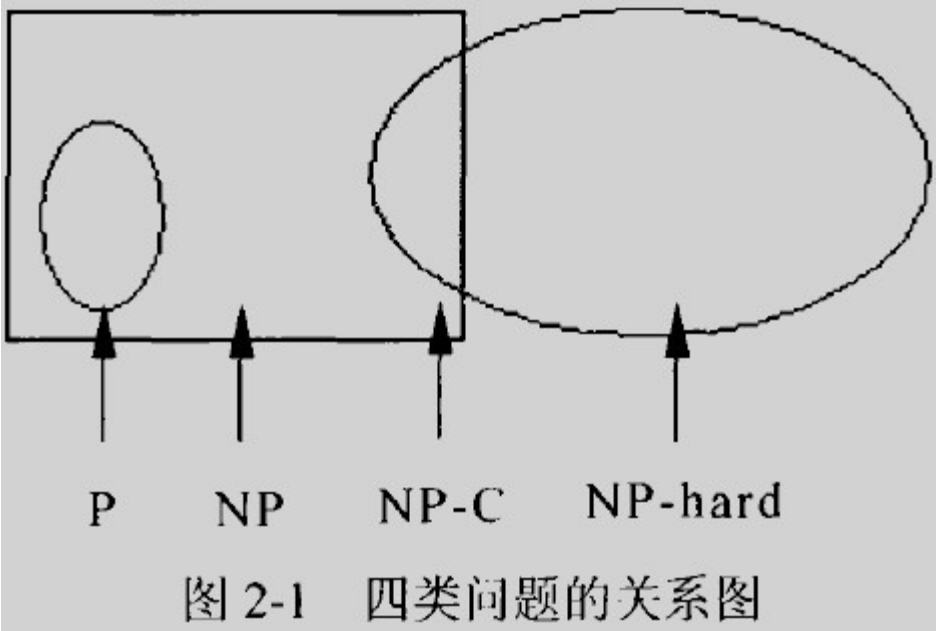


图 2-1　四类问题的关系图

很显然，**所有的P类问题都是NP问题。**也就是说，能多项式地解决一个问题，必然能多项式地验证一个问题的解——既然正解都出来了，验证任意给定的解也只需要比较一下就可以了。关键是，人们想知道，是否所有的NP问题都是P类问题。我们可以再用集合的观点来说明。如果把所有P类问题归为一个集合P中，把所有NP问题划进另一个集合NP中，那么，显然有**P属于NP**。现在，所有对NP问题的研究都集中在一个问题上，即究竟**是否有P=NP？**通常所谓的“NP问题”，其实就一句话：证明或推翻P=NP。

10、讲了这么多的 P 类问题和 NP 类问题，都是理论上的说明，但是实际上，我们经常会碰到一类叫做 NPC 的问题，这个又是什么鬼呢？首先，NPC 是 NP 当中比较特殊的一类问题，它是一些问题的老大，就是说有一些 NP 问题，可以简约成 NPC 问题，也就是说，如果某类 NPC 问题解决了，也可以解决某类的 NP 问题。上截图说明：

时间和精力也没有解决的终极问题，好比物理学中的大统一和数学中的歌德巴赫猜想等。目前为止这个问题还“啃不动”。但是，一个总的趋势、一个大方向是有的。人们普遍认为，P=NP不成立，也就是说，多数人相信，存在至少一个不可能有多项式级时间复杂度的算法的NP问题。人们如此坚信P≠NP是有原因的，就是在研究NP问题的过程中找出了一类非常特殊的NP问题叫做**NP-完全问题**，也即所谓的NPC问题。C是英文单词“完全”（complete）的第一个字母。正是NPC问题的存在，使人们相信P≠NP。下文将花大量篇幅介绍NPC问题，你从中可以体会到NPC问题使P=NP变得多么不可思议。为了说明NPC问题，我们先引入一个概念——**约化**（Reducibility，有的资料上叫“归约”）。简单地说，一个问题A可以约化为问题B的含义即是，**可以用问题B的解法解决问题A，或者说，问题A可以“变成”问题B**。《算法导论》上举了这么个例子。比如说，现在有两个问题：求解一个一元一次方程和求解一个一元二次方程。那么我们说，**前者可以约化为后者**，意即知道如何解一个一元二次方程那么一定能解出一元一次方程。我们可以写出两个程序分别对应两个问题，那么我们能找到一个“规则”，按照这个规则把解一元一次方程程序的输入数据变一下，用在解一元二次方程的程序上，两个程序总能得到一样的结果。这个规则即是：两个方程的对应项系数不变，一元二次方程的二次项系数为0。按照这个

或者可以用其他的话来说，B 我都能搞定，还搞定不了你 A 的？换句话说，B 应该比 A 更复杂。上截图说明：

“问题A可约化为问题B”有一个重要的直观意义：B的时间复杂度高于或者等于A的时间复杂度。也就是说，问题A不比题B难。这很容易理解。既然问题A能用问题B来解决，倘若B的时间复杂度比A的时间复杂度还低了，那A的算法就可以改进为B的算法，两者的时间复杂度还是相同。正如解一元二次方程比解一元一次方程难，因为解决前者的方法可以用来解决后者。

继续 NPC 类问题和 NP 类问题的区别，上截图：

好了，从约化的定义中我们看到，一个问题约化为另一个问题，时间复杂度增加了，问题的应用范围也增大了。通过对某些问题的不断约化，我们能够不断寻找复杂度更高，但应用范围更广的算法来代替复杂度虽然低，但只能用于很小的一类问题的算法。再回想前面讲的P和NP问题，联想起约化的传递性，自然地，我们会想问，如果不断地约化上去，不断找到能“通吃”若干小NP问题的一个稍复杂的大NP问题，那么最后是否有可能找到一个时间复杂度最高，并且能“通吃”所有的NP问题的这样一个超级NP问题？答案居然是**肯定的**。也就是说，**存在这样一个NP问题，所有的NP问题都可以约化成它**。换句话说，只要解决了这个问题，那么所有的NP问题都解决了。这种问题的存在难以置信，并且更加不可思议的是，这种问题不只一个，它有很多个，它是一类问题。这一类问题就是传说中的NPC问题，也就是NP-完全问题。NPC问题的出现使整个NP问题的研究得到了飞跃式的发展。我们有理由相信，NPC问题是最复杂的问题。再次回到全文开头，我们可以看到，人们想表达一个问题不存在多项式的高效算法时应该说它“属于NPC问题”。此时，我的目的终于达到了，我已经把NP问题和NPC问题区别开了。到此为止，本文已经写了近5000字了，我佩服你还能看到这里来，同时也佩服一下自己能写到这里来。

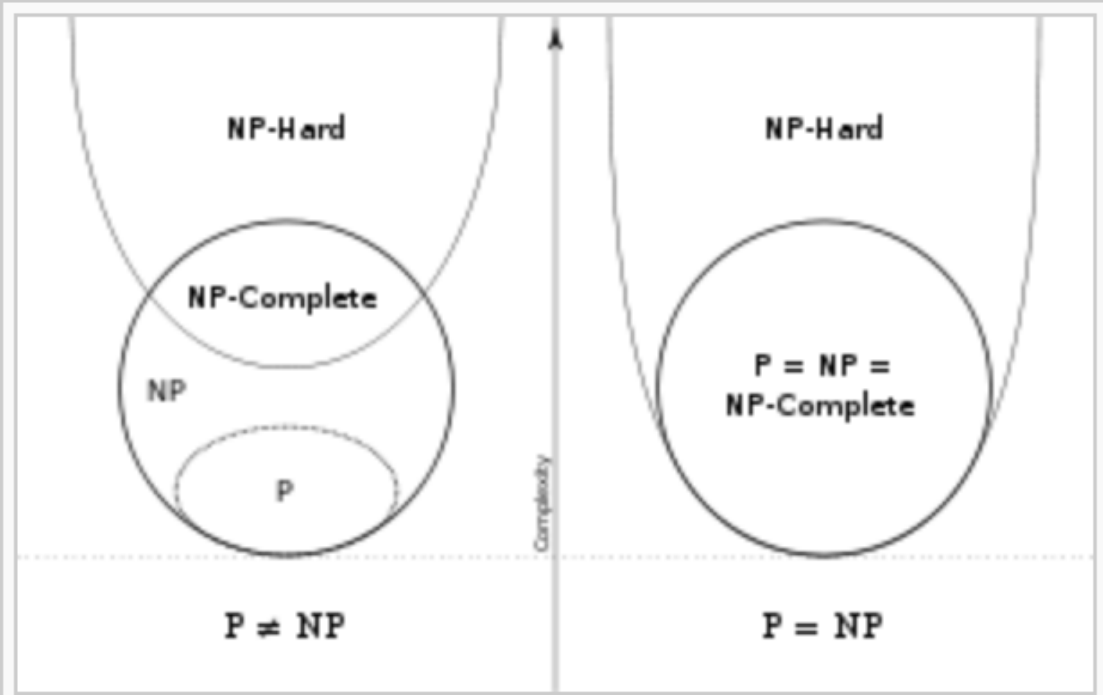
从截图中，我们可以看到：其实 NP 问题有很多类，它们目前都找不到有效的算法求解，但是可以去验证；这些类的 NP 类问题，都可以简约成相关的 NPC 类问题，这些 NPC 类问题都找不到“多项式算法”求解（时间复杂度很高，都是指数级别的或者阶乘级别的）。因此，我们一般说的问题都是说属于 NPC 问题。

11、NPC 类问题清楚了，但是看书的时候，还是会多一个 NPH 的问题，或者说是 NP-Hard 问题。这又是什么鬼？这个和前面的关

系不是很大，但是简单理解一下，NP-Hard 问题不一定是 NP 问题（有可能是），但是它也找不到多项式的算法求解。直接上截图：

顺便讲一下NP-Hard问题。**NP-Hard问题**是这样一种问题，它满足NPC问题定义的第二条但不一定要满足第一条（就是说，NP-Hard问题要比NPC问题的范围广，它不一定是NP问题，读者请尝试回想NP问题和P问题的定义）。NP-Hard问题同样难以找到多项式的算法，但它不列入我们的研究范围，因为它不一定是NP问题。即使NPC问题发现了多项式级的算法，NP-Hard问题有可能仍然无法得到多项式级的算法。事实上，由于NP-Hard放宽了限定条件，它将有可能比所有的NPC问题的时间复杂度更高从而更难以解决。不要以为NPC问题是一纸空谈。NPC问题是存在的。确实有这么一个非常具体的问题属于NPC问题。下文即将介绍它。

因此，NP-Hard 问题不好理解，反正是比 NPC 更复杂的问题吧，他们之间是交集的关系（好像应该是包含的关系吧）。有一些问题既可以是 NPC 问题，也可以是 NP-Hard 问题（或者说 NP-Hard 问题当中，有一部分是 NPC 的问题）。看截图：



Euler diagram for P, NP, NP-complete, and NP-hard set of problems. The existence of problems within **NP** but outside both **P** and **NP**-complete, under the P≠NP assumption, was established by Ladner.[1]

旅行推销员问题是数图论中最著名的问题之一，即“已给一个n个点的完全图，每条边都有一个长度，求总长度最短的经过每个顶点正好一次的封闭回路”。Edmonds，Cook和Karp等人发现，这批难题有一个值得注意的性质，对其中一个问题存在有效算法时，每个问题都会有有效算法。

迄今为止，这类问题中没有一个找到有效算法。倾向于接受NP完全问题（NP-Complet或NPC）和NP难题（NP-Hard或NPH）不存在有效算法这一猜想，认为这类问题的大型实例不能用精确算法求解，必须寻求这类问题的有效的近似算法。

此类问题中，经典的还有 子集和问题； Hamilton回路问题；最大团问题

对 NPC 和 NP-Hard 的问题，看样子只能用搜索算法去得到近似解了。

12、又看到了 NP-Hard 问题的解释。NP-Hard 的问题中，有的时候即使给出了一个解，也不能在有效的时间内判断是否对还是错的。看截图：

六、NP-hard问题

定义：NP-hard问题是这样的问题，只要其中某个问题可以在P时间内解决，那么所有的NP问题就都可以在P时间内解决了。NP-c问题就是NP-hard问题。但注意NP-hard问题它不一定是NP问题，比如，下围棋就是NP-hard问题，但不是NP问题，我们要在一个残局上找一个必胜下法，告诉我们下一步下在哪里。显然，我们找不这个解，而且更难的是，就算有人给我了一个解，我们也无法在P时间内判断它是不是正确的。

上面，主要是看下围棋的这个例子，对于一个残局，你并不知道下一步是否是正确的。因为有三种可能，一种情况是对方无论怎么走，你只要再走正确了，就赢了；一种是对方走正确了，你无论再怎么走都输了；还有就是看对方的情况定输赢。所以，情况复杂到有很多中情况，如果要一个一个的去判断，那又是阶乘级别的问题，非多项式的求解，很复杂。这类问题就是 NP-Hard，而不是 NPC 问题。再看一个截图，来理解一下 NP-Hard 问题。

那么肯定有人要问了，那么NP之外，还有一些连验证解都不能多项式解决的问题呢。这部分问题，就算是NP=P，都不一定能多项式解决，被命名为NP-hard问题。NP-hard太难了，怎样找到一个完美的女朋友就是NP-hard问题。一个NP-hard问题，可以被一个NP完全问题归约到，也就是说，如果有一个NP-hard得到解决，那么所有NP也就都得到解决了。

再看一个截图：

所以对于NP-hard问题，我们可以把他们分成两个部分，一部分可以用polynomial的时间验证一个candidate answer是不是真正的answer，这一部分问题组成了NP-complete集合。

说明，NPC 包含在 NP-Hard 里面。或者可以这里理解 NP-Hard 问题，就是一类好难的问题，跟 NP 问题一样难的问题。确切的说，比 NP 还要难的问题，为什么呢？因为这类 NP-Hard 问题中，简单点的话，就是 NPC 问题，不简单的话，比 NPC 还难。怎么比它还难呢？因为，即使我给出了一个解，你也不能（在多项式的时间里面）判断是否是正确的，还是错误的。看原话的截图：

NP-hard Problem：对于这一类问题，用一句话概括他们的特征就是“at least as hard as the hardest problems in NP Problem”， 就是NP-hard问题至少和NP问题一样难。

2016-12-02

知识需要探讨，才能进步。

1、问题的提出。

经常解决一个问题的时候，会发现有人说这是一个 NP 问题，或者 NPC，还有突然会冒出来 NP-Hard。给人的感觉就是这个问题是个很难的问题，其他的就不知道啦。

2、问题的说明。

还是从定义出发，来说明什么是 P、NP、NPC 和 NPH 问题，但是一定要有案例说明，才能更好的理解。

（1）、P 问题：能在“多项式时间”内解决的问题。看例子：

1）冒泡排序问题。

参考：http://blog.csdn.net/morewindows/article/details/6657829

```
void bubble_sort(int *a, int size)
{
  int i, j, t;
  for(i = 1; i < size; ++i){
    for(j = 0; j < size -i; ++j){
      if(a[j] > a[j+1]){
        t = a[j];
        a[j] = a[j+1];
        a[j+1] = t;
      }
    } // end for j
  }// end for i
}
```

多项式时间是算法复杂度的一个概念。
问题元素为n时，若完成算法的指令条数为n的k次方（k为常数），那么这个算法是多项式时间算法。
比如冒泡排序的时间复杂度是n的平方，就是一个多项式时间算法。

如果指令条数为k的n次方或者n!之类的，就是非多项式时间算法。

2）最短路径问题。

参考：http://blog.chinaunix.net/uid-26548237-id-3834514.html

http://blog.chinaunix.net/uid-26548237-id-3834873.html

最短路径的问题，比较经典，已经用“多项式时间”解决了，因此也是一个 P 问题。

```
/* Dijkstra算法，求有向网g的v0顶点到其余顶点v的最短路径P[v]及带权长度D[v] */
/* P[v]的值为前驱顶点下标,D[v]表示v0到v的最短路径长度和 */
void ShortestPath_Dijkstra(Mgraph g, int v0, Patharc *P, ShortPathTable *D)
{
    int v,w,k,min;
    int final[MAXVEX];                    /* final[w]=1表示求得顶点v0至vw的最短路径 */

    /* 初始化数据 */
    for(v=0; v<g.numVertexes; v++)
    {
        final[v] = 0;                     /* 全部顶点初始化为未知最短路径状态 */
        (*D)[v] = g.arc[v0][v];             /* 将与v0点有连线的顶点加上权值 */
        (*P)[v] = 0;                    /* 初始化路径数组P为0 */
    }

    (*D)[v0] = 0;                         /* v0至v0路径为0 */
    final[v0] = 1;                         /* v0至v0不需要求路径 */

    /* 开始主循环，每次求得v0到某个v顶点的最短路径 */
    for(v=1; v<g.numVertexes; v++)
    {
        min=INFINITY;                     /* 当前所知离v0顶点的最近距离 */
        for(w=0; w<g.numVertexes; w++) /* 寻找离v0最近的顶点 */
        {
            if(!final[w] && (*D)[w]<min)
            {
                k=w;
                min = (*D)[w];            /* w顶点离v0顶点更近 */
            }
        }
        final[k] = 1;                     /* 将目前找到的最近的顶点置为1 */

        /* 修正当前最短路径及距离 */
        for(w=0; w<g.numVertexes; w++)
        {
            /* 如果经过v顶点的路径比现在这条路径的长度短的话 */
            if(!final[w] && (min+g.arc[k][w]<(*D)[w]))
            {
                /* 说明找到了更短的路径，修改D[w]和P[w] */
                (*D)[w] = min + g.arc[k][w]; /* 修改当前路径长度 */
                (*P)[w]=k;
            }
        }
    }
}
```

```
/* Floyd算法，求网图G中各顶点v到其余顶点w的最短路径P[v][w]及带权长度D[v][w]。 */
void ShortestPath_Floyd(MGraph G, Patharc *P, ShortPathTable *D)
{
    int v,w,k;
    for(v=0; v<G.numVertexes; ++v)            /* 初始化D与P */
    {
        for(w=0; w<G.numVertexes; ++w)
        {
            (*D)[v][w]=G.arc[v][w];            /* D[v][w]值即为对应点间的权值 */
            (*P)[v][w]=w;                     /* 初始化P */
        }
    }
    for(k=0; k<G.numVertexes; ++k)
    {
        for(v=0; v<G.numVertexes; ++v)
        {
            for(w=0; w<G.numVertexes; ++w)
            {
                if ((*D)[v][w]>(*D)[v][k]+(*D)[k][w])
                {
                    /* 如果经过下标为k顶点路径比原两点间路径更短 */

                    (*D)[v][w]=(*D)[v][k]+(*D)[k][w];    /* 将当前两点间权值设为更小的一个 */
                    (*P)[v][w]=(*P)[v][k];               /* 路径设置为经过下标为k的顶点 */
                }
            }
        }
    }
}
```

可以从循环可以看出，Dijkstra 算法是二重循环，也就是多项式时间，时间复杂度是 O(n^2)。如果按照此算法求每个点的话，理论上就在外面加了一重循环，因此时间复杂度是 O(n^3)。但是我们一般算每个定点的最短距离时，用的是 Floyd 算法，这是我们可以看到，它是三重循环，时间复杂度也是 O(n^3)。所以它们都是 P 问题。

（2）、NP 问题：可以在“多项式时间”内判断这个答案是否正确。

怎么问题变成了“判定问题”呢？解释一下：如果有些问题，我找不到一个“多项式时间”算法求解，但是我可以针对一个解，我可以用“多项式时间”来判定它是否准确，那么它属于 NP 问题。但是问题来了，我都不知道怎么解是哪个，怎么知道随便拿来的解是对的还是错误的。比如，最短路径问题，我随便找了一条路

径，我怎么知道是不是最短的呢？因此，NP 问题的定义是针对“判定型问题”来说的，这里需要转换一下思路，把上述最短路径的问题变一下，变成找到一条小于 50 米长的最短路径。OK，我不知道怎么找，但是我可以验证一条路径是否小于 50，只要你把路径告诉我，然后，我进行相加，就可以了。相加，那就是进行一个 for 循环，把路段进行相加，时间复杂度 O(n)，n 代表路段的数量。那有人说了，按照你的这种定义，那么求最短路径的问题，不就成了 NP 问题，它本来应该是 P 问题的啊。对的，按照定义来说，它也是 NP 问题，因此所有的 P 问题都是 NP 问题。不用混乱。

那为什么有人说，这个问题是 NP 问题，好难的，求不出来。确切的说，P 问题是 NP 问题的一个子集，有一类 NP 问题，是非常难的，现在不知道用一个“多项式时间”算法求解，但是可以去用“多项式时间”去验证“判定问题（转换问题）”的正确性。这类问题就是 NPC 问题，如果这类问题一旦解决了（用“多项式时间”），所有的 NP 问题就解决了。现实中，这样的例子有背包问题（Knapsack Problem），TSP 问题（Travelling Salesman Problem）等。

1）、背包问题的描述：给定一组 N 物品，每种物品都有自己的重量 W 和价格 V，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。转变成“判定问题”：是否存在这样的组合，在总重量不超过 W 的前提下，总价值是否能达到 V？

2）、TSP 问题的描述：假设有一个旅行商人要拜访 n 个城市，他

必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。转变成“判定问题”：是否存在这样的路径，总里程小于 L?

（3）、NPC 问题：首先是 NP 问题，第二是其他的一些 NP 问题（出了 P 问题）都能“简约”到 NPC 问题。

简单说，NPC 问题比较有代表型的问题，如果我这种代表型问题能够用“多项式时间”算法求解了，那么其他的 NP 问题也就解决了。因此，我们大部分谈论的比较难的问题，或者说这是个 NP 问题，就是指这一类 NPC 问题。所以，我们一般可以记下来这些有代表的 NPC 问题，有哪些呢？下面列举一些著名的 NPC 问题（用 decision problems 来表示的）。

参考：

https://en.wikipedia.org/wiki/NP-completeness#Formal_definition_of_NP-completeness

- Boolean satisfiability problem (SAT)
- Knapsack problem
- Hamiltonian path problem
- Travelling salesman problem (decision version)
- Subgraph isomorphism problem
- Subset sum problem
- Clique problem

- Vertex cover problem
- Independent set problem
- Dominating set problem
- Graph coloring problem

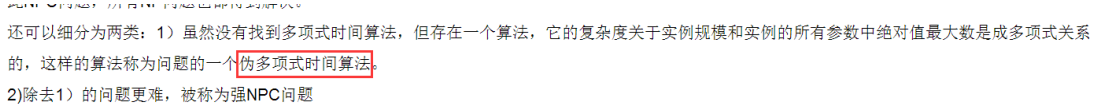
以上的这些问题，到现在都不能用“多项式时间”算法求解。

1)、但是，我们知道的一些问题，不都是有一些求解的方法了吗，比如 TSP 问题？是的，那些方法求出来的是近似的解，而不是精确的解。

2)、还有的问题，好像也是可以求解了，比如背包问题，用动态规划或者分支定界等求解到精确解了。这类问题，随着 n 的增大，是指数级别的增长（虽然看上去不是），这种叫做伪多项式时间。

参考：

http://blog.csdn.net/zephyr_be_brave/article/details/13168555


还可以细分为两类：1）虽然没有找到多项式时间算法，但存在一个算法，它的复杂度关于实例规模和实例的所有参数中绝对值最大数是成多项式关系的，这样的算法称为问题的一个伪多项式时间算法。
2)除去1）的问题更难，被称为强NPC问题

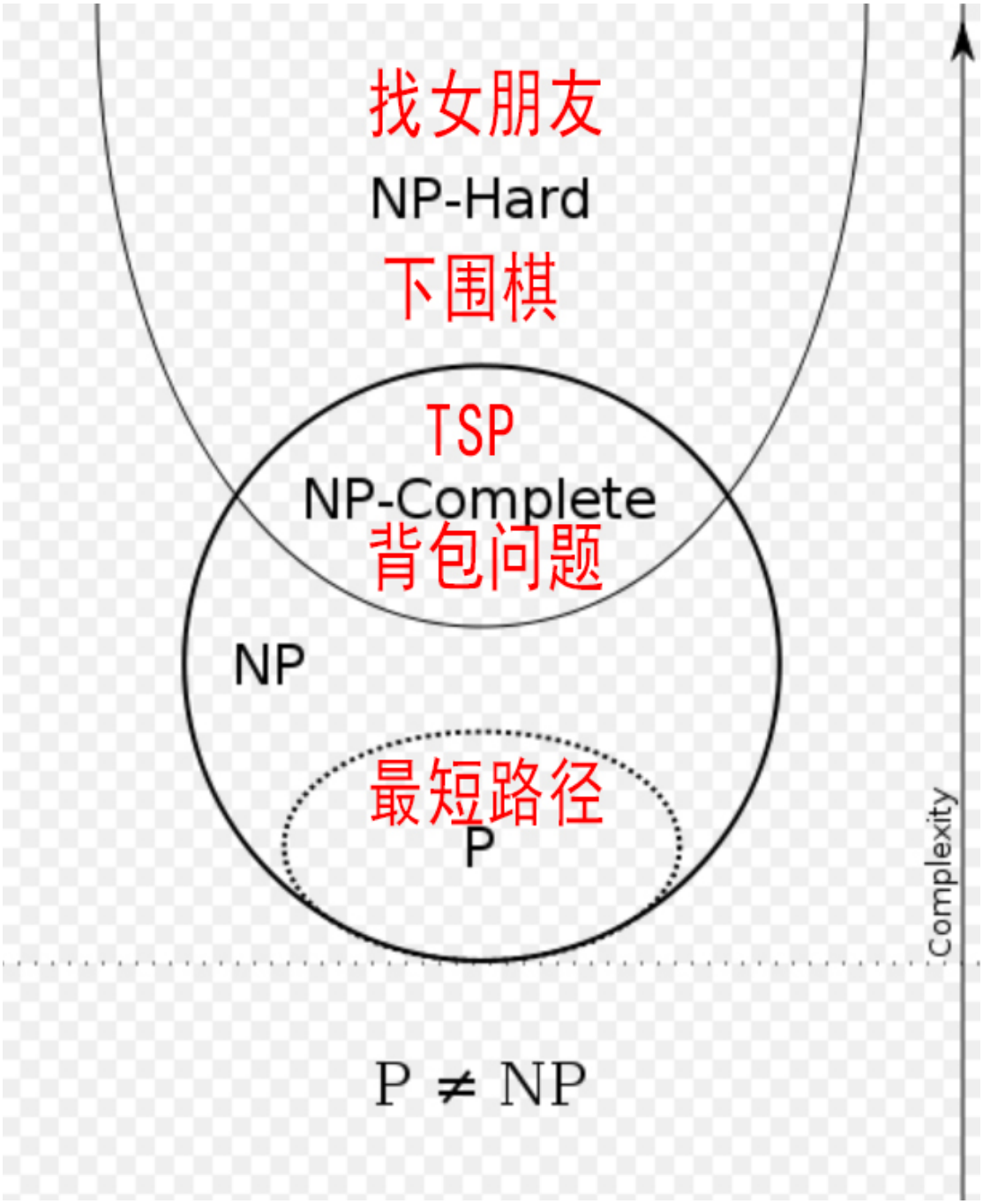
（4）、NPH 问题：at least as hard as the hardest problems in NP，翻译过来，至少像 NP 问题中最难的问题一样难。

NP-Hard 问题，有些问题是 NPC 问题，有些问题比 NPC 还难。比 NPC 还难，就是说这些问题都不能在“多项式时间”内去判断是够正确。有没有，当然有，比如下围棋，你不能判断，是否“这一步”会赢。大致理解一下就 OK 了，因为后面的变数太多了，就是说情况太复杂了，总共有 324 个格子，不知道是指数级别的，

还是阶乘级别的，太复杂了。

3、问题的总结。

简单看，问题可以在一张图上，如下图：



所以，我们一般来说，碰到一个比较困难的问题，我们都说这是一个 NPC 问题，只要是 NPC 问题，那么它也是 NP-Hard 问题。

问题提出：整数规划是不是 NP-Hard 问题？