**特點**

**遺傳演算法在解決最佳化問題過程中有如下特點：**

* **遺傳演算法在適應度函式選擇不當的情況下有可能收斂於局部最優，而不能達到全局最優。**
* **初始種群的數量很重要，如果初始種群數量過多，演算法會占用大量系統資源；如果初始種群數量過少，演算法很可能忽略掉最優解。**
* **對於每個解，一般根據實際情況進行編碼，這樣有利於編寫變異函式和適應度函式(Fitness Function)。**
* **在編碼過的遺傳演算法中，每次變異的編碼長度也影響到遺傳演算法的效率。如果變異代碼長度過長，變異的多樣性會受到限制；如果變異代碼過短，變異的效率會非常低下，選擇適當的變異長度是提高效率的關鍵。**
* **變異率也是一個重要的參數。**
* **對於動態資料，用遺傳演算法求最優解比較困難，因為染色體種群很可能過早地收斂，而對以後變化了的資料不再產生變化。對於這個問題，研究者提出了一些方法增加基因的多樣性，從而防止過早的收斂。其中一種是所謂觸發式超級變異，就是當染色體群體的品質下降（彼此的區別減少）時增加變異機率；另一種叫隨機外來染色體，是偶爾加入一些全新的隨機生成的染色體個體，從而增加染色體多樣性。**
* **選擇過程很重要，但交叉和變異的重要性存在爭議。一種觀點認為交叉比變異更重要，因為變異僅僅是保證不遺失某些可能的解；而另一種觀點則認為交叉過程的作用只不過是在種群中推廣變異過程所造成的更新，對於初期的種群來說，交叉幾乎等效於一個非常大的變異率，而這麼大的變異很可能影響進化過程。**
* **遺傳演算法很快就能找到良好的解，即使是在很複雜的解空間中。**
* **遺傳演算法並不一定總是最好的最佳化策略，最佳化問題要具體情況具體分析。所以在使用遺傳演算法的同時，也可以嘗試其他演算法，互相補充，甚至根本不用遺傳演算法。**
* **遺傳演算法不能解決那些「大海撈針」的問題，所謂「大海撈針」問題就是沒有一個確切的適應度函式表徵個體好壞的問題，使得演算法的進化失去導向。**
* **對於任何一個具體的最佳化問題，調節遺傳演算法的參數可能會有利於更好更快收斂，這些參數包括個體數目、交叉率和變異率。例如太大的變異率會導致遺失最優解，而過小的變異率會導致演算法過早的收斂於局部最優點。對於這些參數的選擇，現在還沒有實用的上下限。**
* **適應度函式對於演算法的速度和效果也很重要。**

**變數**

**最簡單的遺傳演算法將染色體表示為一個**[**數位**](https://zh.wikipedia.org/wiki/%E4%BD%8D)**串，數值變數也可以表示成**[**整數**](https://zh.wikipedia.org/wiki/%E6%95%B4%E6%95%B0)**，或者**[**實數**](https://zh.wikipedia.org/wiki/%E5%AE%9E%E6%95%B0)**（**[**浮點數**](https://zh.wikipedia.org/wiki/%E6%B5%AE%E7%82%B9%E6%95%B0)**）。演算法中的雜交和突變都是在位元組串上進行的，所以所謂的整數或者實數表示也一定要轉化為數位形式。例如一個變數的形式是實數，其範圍是0～1，而要求的精度是0.001，那麼可以用10個數位表示：0000000000表示0，1111111111表示1。那麼0110001110就代表0.398。**

**在遺傳演算法里，**[**精英選擇**](https://zh.wikipedia.org/w/index.php?title=%E7%B2%BE%E8%8B%B1%E9%80%89%E6%8B%A9&action=edit&redlink=1)**是一種非常成功的產生新個體的策略，它是把最好的若干個個體作為**[**精英**](https://zh.wikipedia.org/wiki/%E7%B2%BE%E8%8B%B1%E6%94%BF%E6%B2%BB)**直接帶入下一代個體中，而不經過任何改變。**

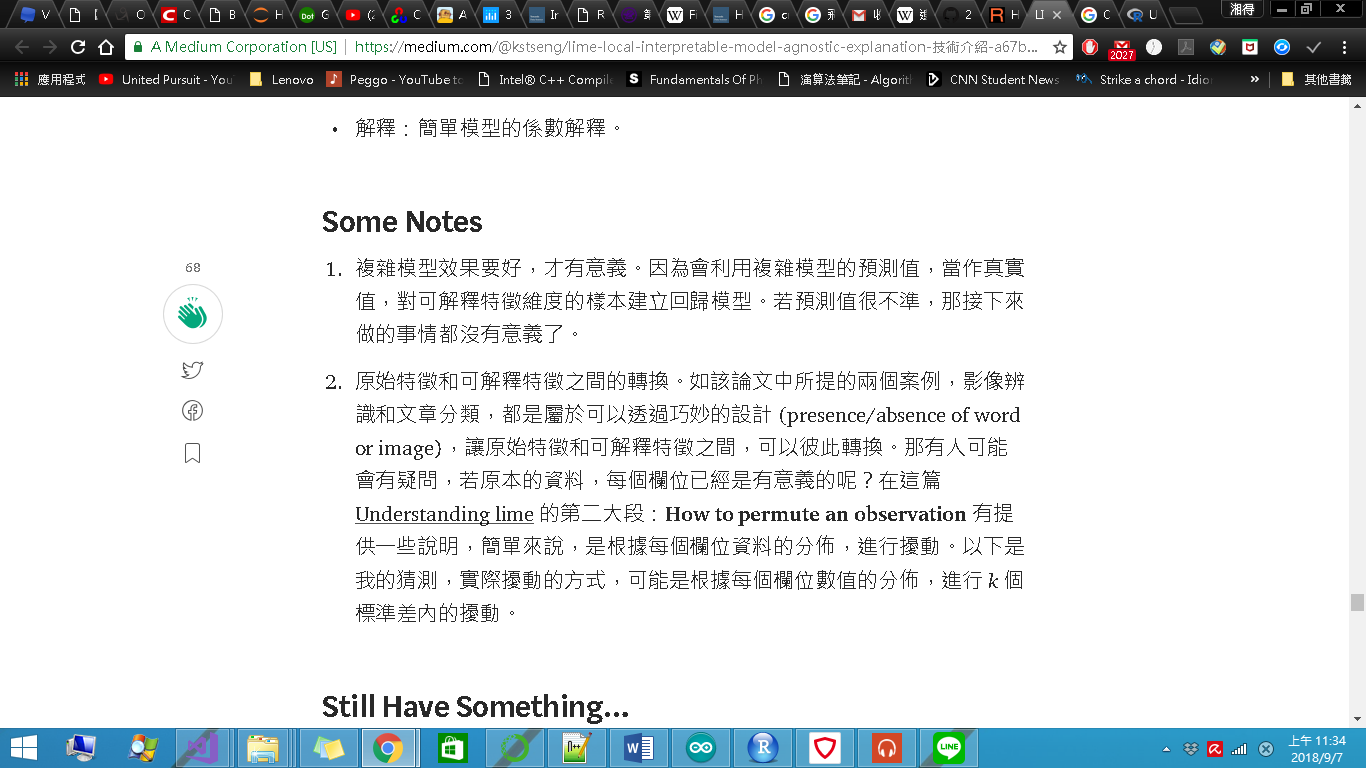
**通過**[**平行計算**](https://zh.wikipedia.org/wiki/%E5%B9%B6%E8%A1%8C%E8%AE%A1%E7%AE%97)**實現遺傳演算法一般有兩種，一種是所謂粗糙並列遺傳演算法，即一個計算單元包含一個種群；而另一種是所謂精細並列遺傳演算法，每一個計算單元處理一個染色體個體。**

**遺傳演算法有時候還引入其他變數，例如在實時最佳化問題中，可以在適應度函式中引入時間相關性和干擾。**

**適用的問題**

**遺傳演算法擅長解決的問題是**[**全局最佳化問題**](https://zh.wikipedia.org/wiki/%E7%BB%84%E5%90%88%E6%9C%80%E4%BC%98%E5%8C%96)**，例如，解決**[**時間表安排**](https://zh.wikipedia.org/w/index.php?title=%E6%97%B6%E9%97%B4%E8%A1%A8%E5%AE%89%E6%8E%92&action=edit&redlink=1)**問題就是它的一個特長，很多安排時間表的軟體都使用遺傳演算法，遺傳演算法還經常被用於解決實際**[**工程問題**](https://zh.wikipedia.org/w/index.php?title=%E5%B7%A5%E7%A8%8B%E9%97%AE%E9%A2%98&action=edit&redlink=1)**。**

**跟傳統的**[**爬山演算法**](https://zh.wikipedia.org/wiki/%E7%88%AC%E5%B1%B1%E7%AE%97%E6%B3%95)**相比，遺傳演算法能夠跳出局部最優而找到全局最優點。而且遺傳演算法允許使用非常複雜的適應度函式（或者叫做**[**目標函式**](https://zh.wikipedia.org/w/index.php?title=%E7%9B%AE%E6%A0%87%E5%87%BD%E6%95%B0&action=edit&redlink=1)**），並對變數的變化範圍可以加以限制。而如果是傳統的爬山演算法，對變數範圍進行限制意味著複雜的多的解決過程，這方面的介紹可以參看**[**受限最佳化問題**](https://zh.wikipedia.org/w/index.php?title=%E5%8F%97%E9%99%90%E4%BC%98%E5%8C%96%E9%97%AE%E9%A2%98&action=edit&redlink=1)**和**[**非受限最佳化問題**](https://zh.wikipedia.org/w/index.php?title=%E9%9D%9E%E5%8F%97%E9%99%90%E4%BC%98%E5%8C%96%E9%97%AE%E9%A2%98&action=edit&redlink=1)**。**



**幸運時只要攪1次就可成功**