爬山法和模擬退火算法通常用來求解 TSP 的最短路徑問題。爬山法的一個最大的缺點就是，它只能獲取一個局部最優的解，但是無法獲取一個全局最優的解。而模擬退火算法，它以一定的機率接受較差的解，因此，可以在一定程度上避免局部最優的問題。而迪傑斯特拉算法雖然能夠得到最短路徑，但是由於需要大量的計算，比較消耗性能，因此實際應用中並不多。

局部搜索是解決**最優化問題**的一種**啟發式算法**。對於某些計算起來非常複雜的最優化問題，比如各種 NP 完全問題，要找到最優解需要的時間隨問題規模呈指數增長，因此誕生了各種啟發式算法來退而求其次尋找次優解，是一種近似算法（Approximate algorithms），以時間換精度的思想。局部搜索就是其中的一種方法。

對於**組合問題**，給出如下定義：

Definition 1.1. *A* **Combinatorial Optimization** *problem* $P = (S, f)$ *can be defined by:*

- *a set of variables* $X = \{x_1, \ldots, x_n\}$;
- *variable domains* $D_1, \ldots, D_n$;
- *constraints among variables;*
- *an objective function* f *to be minimized,*[1] *where* $f : D_1 \times \ldots \times D_n \to \mathbb{R}^+$;

The set of all possible feasible assignments is

$$S = \{s = \{(x_1, v_1), \ldots, (x_n, v_n)\} \mid v_i \in D_i, s \text{ satisfies all the constraints}\} .$$

其中，S 為搜索空間（解空間），其中的每一元素都是問題一個可能解。解決組合問題，即是找到一個 s* ∈ S，使得目標函數 f 值最小。s*稱為全局最優解。

對於**鄰域動作**定義如下：

Definition 1.2. *A* **neighborhood structure** *is a function* $\mathcal{N} : S \to 2^S$ *that assigns to every* $s \in S$ *a set of neighbors* $\mathcal{N}(s) \subseteq S$. $\mathcal{N}(s)$ *is called the neighborhood of* s.

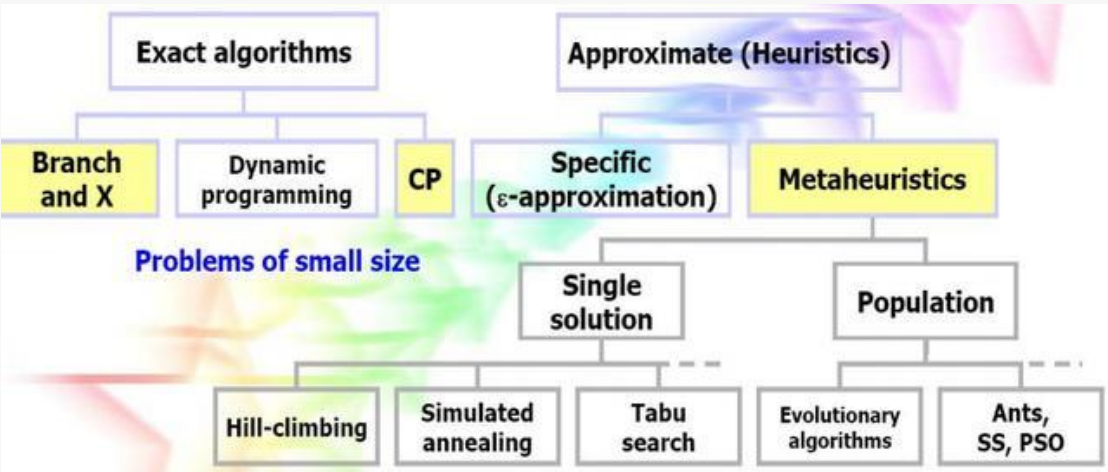
鄰域動作是一個函數，通過這個函數，對當前解 s，產生其相應的鄰居解集合。例如：對於一個 bool 型問題，其當前解為：s = 1001，當將鄰域動作定義為翻轉其中一個 bit 時，得到的鄰居解的集合 N(s)={0001,1101,1011,1000}，其中 N(s) ∈ S。同理，當將鄰域動作定義為互換相鄰 bit 時，得到的鄰居解的集合 N(s)={0101,1001,1010}.

2、過程描述

局部搜索算法從一個初始解開始，通過鄰域動作，產生其鄰居解，判斷鄰居解的質量，根據某種策略，來選擇鄰居解，重複上述過程，至到達終止條件。不同局部搜索算法的區別就在於：鄰域動作的定義和選擇鄰居解的策略，也是決定算法好壞的關鍵（**集中性**和**發散性**，Intensification and Diversification）。

3、算法簡介

對於優化問題相關算法有如下分類：



下文分別簡單介紹幾個局部搜索相關算法，也是基於個體的啟發式算法（Single solution）。

3.1 爬山法（HILL-CLIMBING）

爬山法與 Iterative Improvement 的思想是一樣的，區別在於前者尋找最大值，後者尋找最小值。一種完全的貪心的思想，有更好的，則選擇更好的，沒有更好的，則終止。

```
s ← GenerateInitialSolution()
repeat
    s ← Improve(N(s))
until no improvement is possible
```

流程如上圖所示，判斷當前解 s 的鄰居解質量，若其中有比當前解更好的解，則 s = Improve(N(S))，令當前解等於

鄰居解中質量最好的解，重複上述過程，直至鄰居解中沒有更好的解為止。

缺點：很容易陷入局部極值，最終解的好壞與初始解的選擇有很大關係。

我們今天要解決的一個場景就是有 280 個城市，已知他們的二維坐標。求從某個城市出發，遍歷整個城市，並回到出發點的最短路徑。

測試數據從 http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/a280.tsp.gz 網上下載，格式如下：

```
NAME : a280
COMMENT : drilling problem (Ludwig)
TYPE : TSP
DIMENSION: 280
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
  1 288 149
  2 288 129
  3 270 133
  4 256 141
  5 256 157
  6 246 157
  7 236 169
  8 228 169
  9 228 161
 10 220 169
 11 212 169
 12 204 169
 13 196 169
 14 188 169
 15 196 161
 16 188 145
 17 172 145
 18 164 145
 19 156 145
```

算法的思路是：

1、計算所有節點兩兩之間的歐式距離，存放在二維數組中。

2、構建一個空的集合，作為最短路徑的集合。一個值為 0 的變量，作為最初始的最短路徑值。

while 最短路徑集合中的元素個數 <= 所有的節點數

do :

3、構建鄰域。每次從所有節點中，如果原始的節點集合大於或等於 10，則從原始節點集合中隨機選擇所有節點的十分之一個節點作為鄰域。如果小於 10，則用原始節點集合的個數對 10 取模的結果，作為鄰域的節點個數。

4、如果最短路徑的集合為空，則從鄰域中隨機選擇一個。如果不為空，則選取最短路徑集合中的最後一個元素，與鄰域中的元素，依次計算其歐氏距離，選擇距離最小的節點，添加到最短路徑集合中，作為最短路徑集合中的最後一個元素的下一個節點。同時，在原來的最短路徑值的基礎上，加上當前的最短路徑。從原始的節點集合中，則刪除當前選擇的節點。

輸出最短路徑的節點序列及相應的最短路徑值，並在坐標系中畫出該最短路徑

下面，我們開始採用 python 編程實現。關鍵代碼如下：

```
## 計算任意兩點間的距離
num = len(list)
arr = [[ col for col in range(num)] for row in range(num)]
valstr = ""
for row in range(num):
for col in range(num):
if col == row:
arr[row][col] = 0
else:
p1 = list[row]
p2 = list[col]
arr[row][col] = round(math.sqrt(math.pow((p1.x - p2.x),2) +
math.pow((p1.y - p2.y),2)),2) ### 求歐式距離，保留 2 位小數
## print the matrix for check
"""
for row in range(num):
```

```
for col in range(num):
if (col+1)%10 == 0 :
print valstr + "\n"
valstr = ""
valstr += str(arr[row][col]) + ","
"""
print "爬山法查找最短路徑："
###  參數：最小路徑的最後一個節點和鄰域
def valSum(curnode,nextnodeList):
if nextnodeList == None or len(nextnodeList) < 1 :
print "empty"
return 0
mincost = sys.maxint
retnode = 0
for node in nextnodeList:
if arr[curnode][node] < mincost:
mincost = arr[curnode][node]
retnode = node
return (retnode,mincost)
```

```
indexList = [ i for i in range(num)] ###  原始的節點序列
selectedList = [] ##  選擇好的元素
###  具體思想是：  從剩餘的元素中隨機選擇十分之一的元素，作為鄰域。然後從鄰域中選擇一個元素作為已經構建好的最小路徑的下一個節點，使得該路徑
mincost = sys.maxint ###最小的花費
count = 0 ###  計數器
while count < num:
count += 1
###  構建一個鄰域:  如果 indexList 中元素個數大於 10 個，則取樣的個數為剩餘元素個數的十分之一。否則為剩餘元素個數對 10 的取餘數
leftItemNum = len(indexList)
# print "leftItemNum:" ,leftItemNum
nextnum = leftItemNum//10 if leftItemNum >= 10 else
leftItemNum%10
nextnodeList = sample(indexList,nextnum) ###  從剩餘的節點中選出 nextnum 個節點
if len(selectedList) == 0 :
```

```
item = choice(nextnodeList)
selectedList.append(item)
indexList.remove(item)
mincost = 0
continue
curnode = selectedList[len(selectedList) - 1]
# print "nextnodeList:" ,nextnodeList
nextnode, cost = valSum(curnode,nextnodeList) ###  對待選的序列路徑求和
### 將返回的路徑值添加到原來的路徑值上，同時，在剩餘的節點序列中，刪除 nextnode 節點
mincost += cost
indexList.remove(nextnode)
selectedList.append(nextnode)
print "最合適的路徑為：" ,selectedList
print "路徑節點個數：" ,len(selectedList)
print "最小花費為：" , mincost
#################### 求全局最短路徑 #######
print "\n\n 全局最短路徑"
```

```
cost = 0

slist = []

nextList = [ i for i in range(num)] ###  原始的節點序列

finalcost = 0

count = 0

while count < num:

count += 1

if len(slist) == 0 :

item = choice(nextList)

slist.append(item)

nextList.remove(item)

finalcost = 0

continue

curnode = slist[len(slist) - 1]

# print "nextList:" ,nextList

nextnode, cost = valSum(curnode,nextList) ###  對待選的序列路徑求和

# print ("nextnode = %d , cost = %d" %(nextnode,cost))
```

將返回的路徑值添加到原來的路徑值上，同時，在剩餘的節點序列中，刪除 nextnode 節點

finalcost += cost

nextList.remove(nextnode)

slist.append(nextnode)

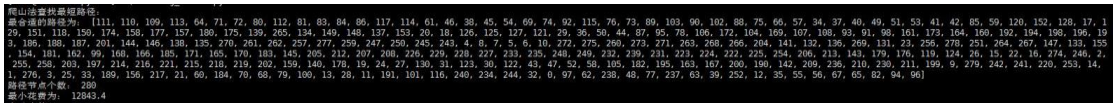
print "finalcost:" ,finalcost, " cost:" ,cost

print "最合適的路徑為：" ,slist

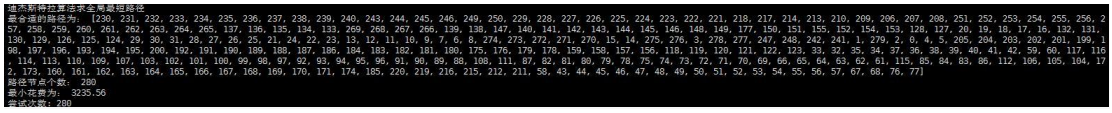
print "路徑節點個數：" ,len(slist)

print "最小花費為：" , finalcost

得到的計算結果為：



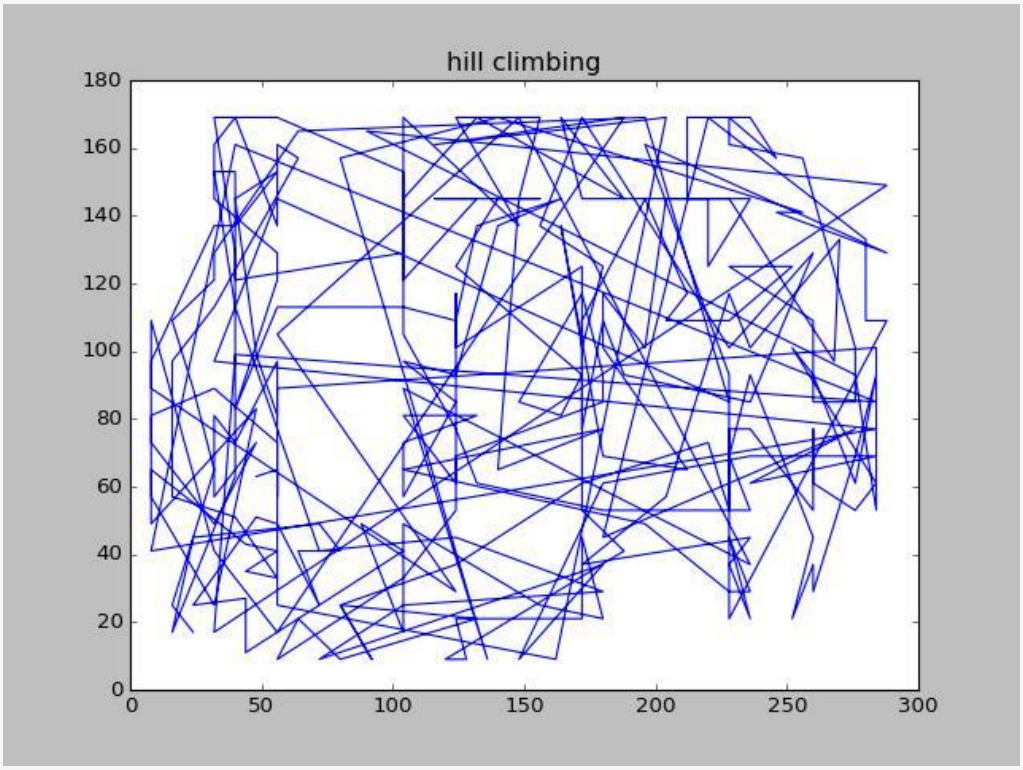
爬山法求解的路徑序列



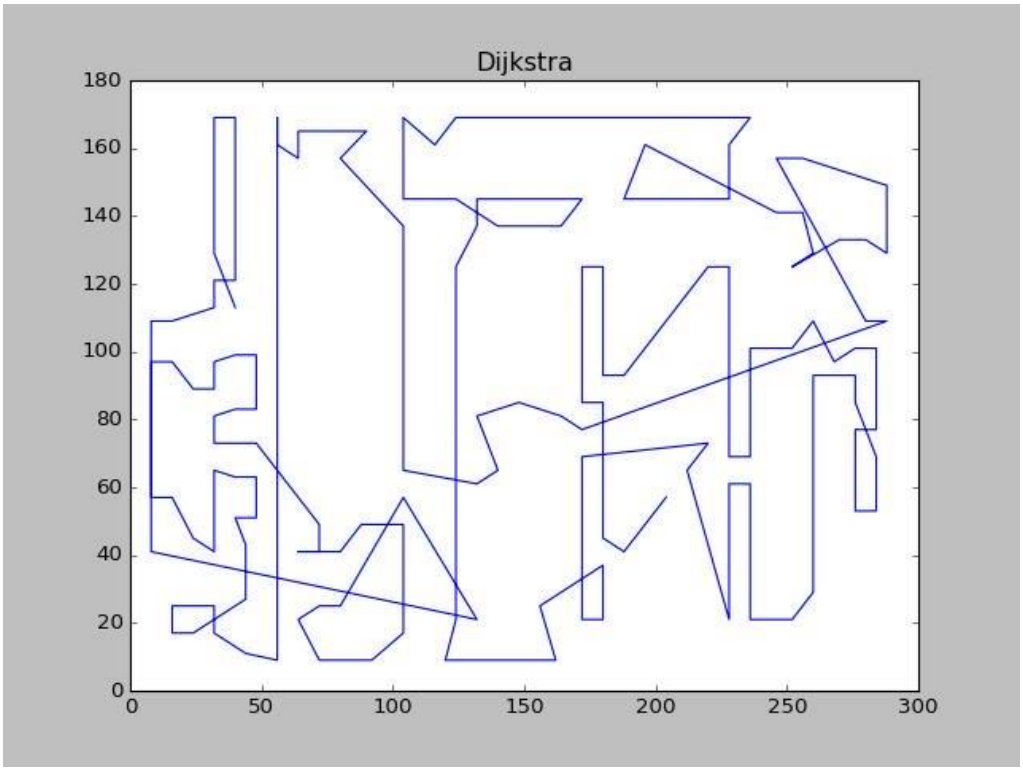
.

迪傑斯特拉算法求解的路徑序列

將得到的結果序列在坐標系中畫出來的效果分別如下：



爬山法得到的最短路徑示意圖



迪傑斯特拉算法得到的最短路徑示意圖

從上可以看出，採用爬山法得到的只是一個局部最優解，與採用迪傑斯特拉算法求得的全局最優解還是有相當的一段距離，但勝在效率高，計算複雜度低，因此，在實際應用中也較多。下一節我們將介紹用 python 實現模擬退火算法的問題。

參考網址: https://www.cnblogs.com/JiePro/p/Metaheuristics_0.html