統計模擬期末報告：以模擬退火法及回溯法解決最佳化問題–以數獨為例。

110354011 涂于珊

摘要：

數獨是一種非常熱門的休閒娛樂，遊戲由9×9的格子組成，玩家需要根據格子已提供的數字推理出剩餘的格子，其中每一列、每一行與每一個3×3的九宮格均須包含1~9，不能缺少也不能重複。

這類型的題目非常考驗電腦或人腦的計算能力，以廣義的角度來說，我們必須在一個不完整的$n^2\times n^2$的格子中優先考慮$n\times n$的解，其屬於 NP-complete 類型的問題，隨著n越大計算的複雜度會以e^n倍的速度成長。那接下來欲透過在組合最佳化問題中有不錯表現的模擬退火法(Simulated Annealing)及回溯法(Backtracking)來比較兩者在此問題上的表現，以正確率及花費的時間為指標，並且實作三種困難程度－Easy、Medium 及 Hard 來觀察其差異。

一、 模擬退火法(Simulated Annealing, SA)

模擬退火演算法（Simulated Annealing, SA）是 S.Kirkpatrick, C. D. Gelatt 和 M.P.Vecchi 等人於 1983 年所提出的通用概率演算法，用來在一個範圍空間內搜尋問題的最佳解，是基於 Monte-Carlo 迭代求解策略的一種隨機尋優算法，其靈感來自於冶煉金屬的降溫過程，於此過程中熱運動趨近於穩定，等同於優化過程中解的收斂。

演算法解析:

- 設定目標函數:計算每一列、每一行及每一九宮格的重複值，目的是最小化

目標函數，使其收斂至 0。

```
target <- function(s){
  tar <- sum(apply(s,1,duplicated)+apply(s,2,duplicated))#前:對列回傳是否為重複值
  for(r in 1:9){#3*3 九宮格是否有重複值
    bloa <- (1:3)+3*(r-1)%%3
    blob <- (1:3)+3*trunc((r-1)/3)
    tar <- tar+sum(duplicated(as.vector(s[bloa,blob])))
  }
  return(tar)
}
```

- 設定迭代次數(Niter)及收斂的速度(Nsteps)，另外透過溫度序列來調整是否接受新解的嚴格程度或說賦予我們「舊的目標函數與新的目標函數間的差值」不同的權重。如下圖的 lcur 為初始權重。

```
lmax <- 10^5
temps <- exp(sqrt(seq(1,log(lmax)^2,le=Niter+1)))#le:切幾分
lcur <- temps[1]#2.718282e+00
```

- 隨機訪查每個格子，如果為空缺則 go through 1~9，先判斷可以放入的數字(即每一行每一列每一九宮格無重複的值)，在判斷哪個是唯一解，是的話填入，否則跳出迴圈。

```
for(t in 1:100){# random order for visit of all sites
  for(i in sample(1:81)){
    if(s[i]==0){
      a=((i-1)%%9)+1
      b=trunc((i-1)/9)+1
      boxa=3*trunc((a-1)/3)+1
      boxa=boxa:(boxa+2)
      boxb=3*trunc((b-1)/3)+1
      boxb=boxb:(boxb+2)
      for (u in (1:9)[pool[a,b,]]){#eliminates impossible values
        pool[a,b,u]=(sum(u==s[a,])+sum(u==s[,b])
                     +sum(u==s[boxa,boxb]))==0
      }
      if (sum(pool[a,b,])==1){ # only one possible case, solution found!
        s[i]=(1:9)[pool[a,b,]]
```

```
            #print(s)
          }
          if (sum(pool[a,b,])==0){ # solution does not exist, exit!
            print("wrong sudoku")#sum(pool[a,b,])==0 可以放 0 種數字
            break()
          }
        }
      }
```

- 接著隨機填入數字，得到目標函數(tarcur)

```
cur <- s
for(r in (1:81)[s==0]){#隨機填數字
  cur[r]=sample((1:9)[pool[r+ 81*(0:8)]],1)
}
tarcur <- target(cur)#重複的數字有幾個
```

再從上一步空缺的值隨機抽樣填入，目標函數為(target(prop))，比較兩者的目標函數差值，透過 log(runif(1))/lcur < tarcur-target(prop)判斷是否接受新的解，由於 log(runif(1))為負值，基本上只有在差值為正時才會接受其解，另外 lcur 會隨著 itertion 越大值越大，因此判斷會越來越嚴苛。

```
for(t in 1:Niter){
  if (tarcur==0){
    print(t)
    print(cur)
    break()
  }
  nchange <- 0
  for(d in 1:Nsteps){
    prop <- cur
i=sample((1:81)[as.vector(s)==0],sample(1:sum(s==0),1,pro=1/(1:sum(s==0))))#
連抽幾個都是隨機抽樣
    for (r in 1:length(i))
      prop[i[r]]=sample((1:9)[pool[i[r]+81*(0:8)]],1)
    if (log(runif(1))/lcur<tarcur-target(prop)){#如果新的比較好就接受新解
      nchange=nchange+(tarcur>target(prop))
      cur=prop#新的解
```

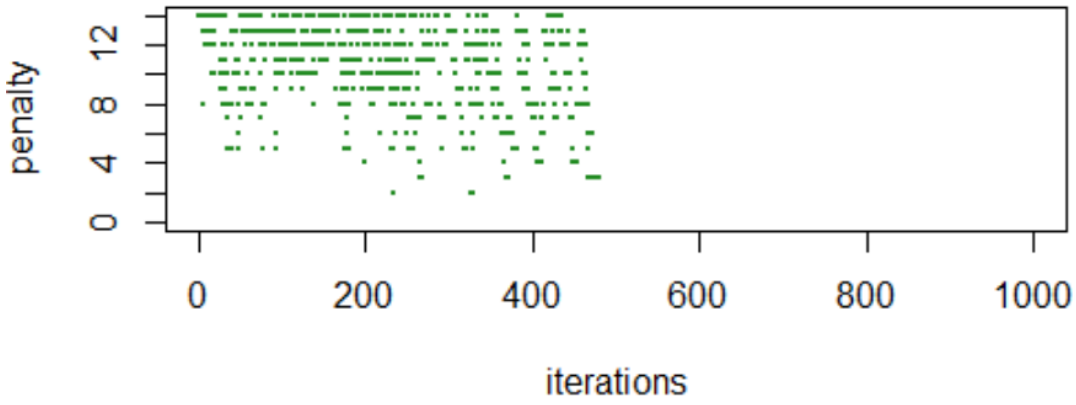
```
        points(t,tarcur,col="forestgreen",cex=.3,pch=19)
        tarcur=target(cur)#新的 tarcur
      }
      if (tarcur==0){
        break()
      }
      lcur=sample(c(1,10^(-
4)),1,pro=c(1.5*(log(t+1))+1,1))*temps[t+1]#15176.3
    }
}
```

實驗結果:

註:由於是隨機塞值，所以每次收斂的結果可能不同。設 set.seed(12345)

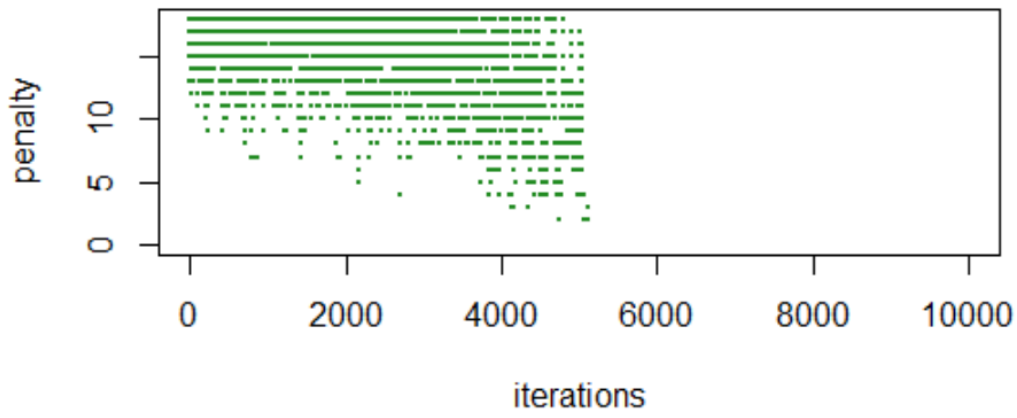
- 簡單: 設定 Niter=1000,Nsteps=10

  直到目標函數收斂至零之迭代次數:480 次



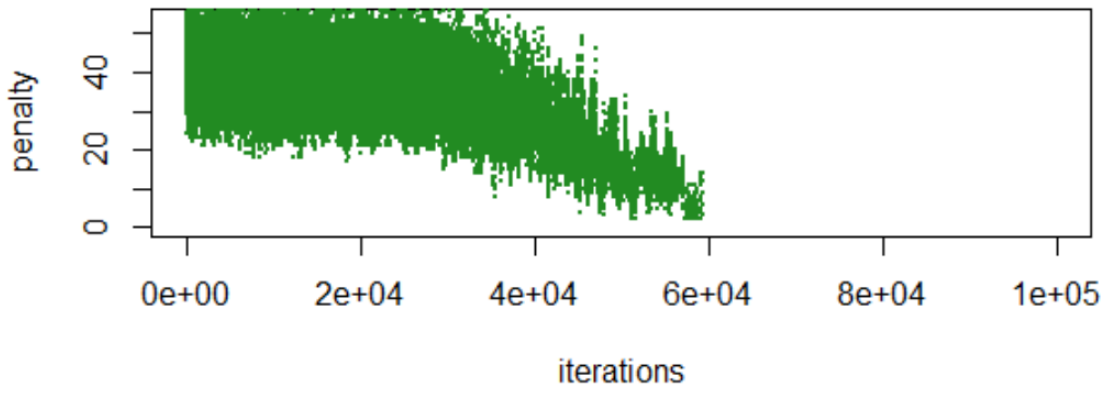
- 中等: 設定 Niter=10000,Nsteps=10

  直到目標函數收斂至零之迭代次數:5117 次



- 難: 設定 Niter=100000,Nsteps=10

  直到目標函數收斂至零之迭代次數:59358 次



二、 回溯法(Backtracking)

回溯法是一種深度優先搜索方法，它會在移動到另一個分支之前完全探索一個分支以找到可能的解決方案。簡單來說就是暴力求解，通過在第一個單元格中放置數字“1”並檢查它是否有違規（檢查行、列和框約束），若無則算法前進到下一個單元格並在該單元格中放置“1”；若有則將該值留空並移回前一個單元格嘗試放入數字“2”。重複此操作，直到發現最後一個（第 81 個）單元格中的允許值。

```
for num in range(1, 10):
        i+=1#計算迭代次數
        print(i)
        # if looks promising
        if(check_location_is_safe(arr,row, col, num)):
            # make tentative assignment
            arr[row][col]= num
            # return, if success,
            # ya !
            if(solve_sudoku(arr,i)):
                return True
            # failure, unmake & try again
            arr[row][col] = 0
```

三、 結論

數獨的資料來源來自 MathWorks 的 Solve and Create SUDOKU puzzles for different levels，裡頭包含不同難度(evil, hard, medium, easy etc)。

Easy: 43 個數字

4	0	0	7	6	0	0	1	2
7	8	2	0	1	5	0	4	9
0	6	1	4	2	8	3	7	0
5	1	0	0	0	4	7	6	3
3	0	0	0	0	0	0	2	8
0	0	6	9	3	0	0	0	0
2	0	0	0	4	0	0	0	0
1	4	7	5	9	3	2	8	6
6	0	0	0	0	0	0	0	0

Medium: 40 個數字

0	0	0	0	0	2	1	0	0
2	0	3	0	1	0	6	9	0
0	0	0	4	3	0	0	8	0
0	0	1	5	0	8	0	4	0
0	0	9	7	0	1	5	6	0
5	7	2	6	0	3	9	1	0
0	0	8	0	0	4	3	0	6
1	2	4	3	0	0	8	5	9
0	6	0	0	0	9	4	0	0

Hard: 32 個數字

0	0	7	0	0	0	0	8	0
0	0	8	0	2	0	9	0	5
0	9	0	0	5	0	1	0	0
0	0	0	4	3	5	7	2	0
7	0	3	0	6	0	4	9	0
0	2	0	0	0	0	5	3	6
0	0	5	6	0	4	2	0	0
2	0	0	0	0	0	0	4	0
0	0	0	2	0	3	8	0	0

Iteration 比較

	Simulated Annealing	Backtracking
Easy	480	204
Medium	5117	215
Hard	59358	249

CPU 計算時間比較

	Simulated Annealing	Backtracking(秒)
Easy	12(秒)	0.020941972732543945
Medium	27(秒)	0.11069822311401367
Hard	40(分)	0.691110372543335

同樣都在約束條件下得到解答的情況下，由 Iteration 及 CPU 計算時間來看很明顯地 Backtracking 的方法比較好。另外 Backtracking 有幾項優點：

- 保證有解決方案
- 解決時間大多與難易程度無關。
- 程序代碼比其他算法(SA)更簡單。

四、 參考資料

數獨資料來源

- https://ww2.mathworks.cn/matlabcentral/fileexchange/13846-solve-and-create-sudoku-puzzles-for-different-levels

Simulated Annealing

- Sudoku via simulated annealing:

  https://xianblog.wordpress.com/2010/02/23/sudoku-via-simulated-annealing/

- Techniques for Solving Sudoku Puzzles

  https://arxiv.org/abs/1203.2295

Backtracking

- Sudoku | Backtracking-7

  https://www.geeksforgeeks.org/sudoku-backtracking-7/