國立臺中教育大學

資訊工程學系

人工智慧期中報告

專題組員：郭承諺

茆明泉

朱皓安

洪　亮

吳政賢

林侑緯

中華民國一○三年四月

一、 前言

PID 控制器是在工業控制上常會需要用到的一個元件，它透過將設定值與系統產生的輸出做比較，再將其誤差提供給 PID 控制器，與 PID 的三個常數 P 和 I 和 D 合作用於計算新的輸入值，使得輸出值能夠與設定值的誤差越來越小，通常人工填整 P 和 I 和 D 十分花費時間，於是我們便想利用基因演算法來達成快速找到最佳的 PID 組合的目標，使其能快速消除誤差，減輕人工調整的負擔。

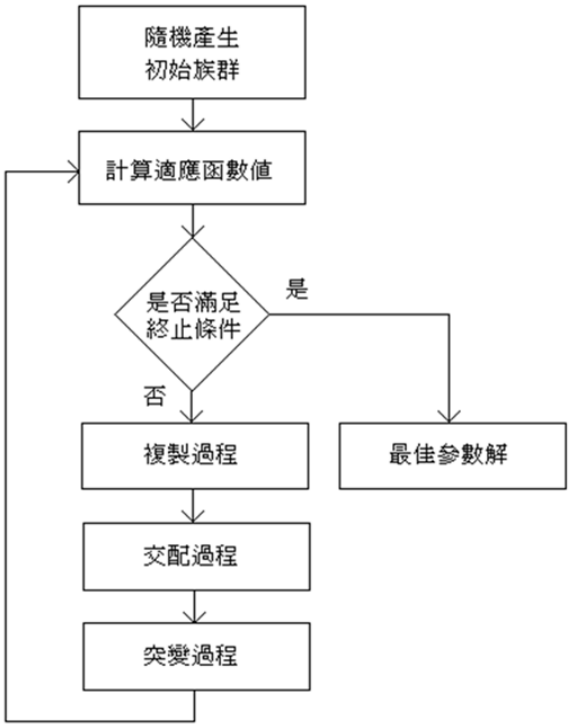
本專題將示範一個利用藉由基因演算法來尋找最佳的 PID 參數值，發光二極體（LED）與光敏電阻（CDs）互相配合來因應環境光（Ambient Light）的強弱來自動調整亮度，達到快速消除系統輸出與預期值的誤差，相較於前人多數使用的轉移函數模擬實驗電路之方法，我們改以實際的硬體電路進行實驗，藉以減少設計轉移函數的步驟。

二、 文獻探討

(一)基因演算法

基因演算法是在 1975 年由 John Holland 所提出，是一種有效的最佳化搜尋方法，近來被廣泛的應用於搜尋各類問題，此演算法最主要的精神在於利用達爾文進化論「物競天擇，適者生存」的方式，藉由每組基因在每代間進行演化，最終找到適應問題的全域最佳解。

基因演算法的幾個重要元件及運作流程包含：初始族群、適應函數、選擇、交配、突變以及結束條件。流程圖如下圖所示。



初始族群

首先必須要先定義基因的編碼方式，在基因演算法的一開始，必須先隨機的產生 N 組基因，N 為族群的總個體數，在運算流程中計算的即為基因，是由一串數字串接而成的字串，每一個基因都對應到目標問題的一個解，基因的長度及個體都會影響到計算目標問題的精確度與複雜度。

適應函數

適應函數用來評估族群中每一個個體適合度的指標，符合進化論中提到「適者生存，不適者淘汰」的基本概念，適合度函數值越高表示個體的適合度越好、競爭力越強，藉由判別基因的適合度函數值指標，進而決定是否保存此染色體。在基因演算法的演化運算過程中，每一代所產生的族群都會有適應性函數值，當數值比上一代組群還高，表示新一代的族群比上一代族群還要優秀，而新一代則群的染色體被保存下來的機會也越高，在經過幾代的演化後，保留下來的族群就是適應性函數最佳的。可以藉由設計不同的適合度函數來達到控制演化的方向，且適應性函數

的設計好壞將會直接影響到是否可得到最後的最佳解。

選擇

選擇是利用基因計算出的適應函數值來決定此基因被保留到下一代的機率，數值越高的個體，被複製成為下一代的個體之機率就相對的越高，較常用的複製方式有以下幾種：

1.競爭式選擇法

2.隨機挑選法

3.輪盤式選擇

交配

被選入交配池後，兩個母代個體會根據交配率彼此交換位元來產生兩個新的個體，期待能從此過程中產生出更優秀的個體。常用的交配方式分為單點、兩點以及均勻三種。

突變

隨機選取一染色體並且選取突變基因，以改變染色體內的資訊

結束條件

基因演算法的最後需要一個終止條件，終止條件分為以下幾種：

(1)演化達到指定的世代數目

(2)演化流程達到要求的目標適應函數值

(3)演化停滯時

(二)PID 控制器與基因演算法

隨著工業技術的不斷發展，各種設備的複雜度也與日俱增，因此也導致了需要控制部分的增加，若是設備所產出的結果和預期的不相同我們就必須要進行調整使得誤差值能夠消失，現今的工業控制常常使用 PID 控制器進行誤差值的修正，PID 控制器使用 3 種演算法來進行誤差

修正，分別是比例(P)、積分(I)、微分(D)，這也是為何它被稱作 PID 控制器的原因，為了使誤差快速收斂，我們需要一組參數(Kp, Ki, Kd)，一般來說這組數值大多是由人工慢慢調整以找出最適合的組合，但這個過程相當耗費時間及人力，所以這次我們想到利用基因演算法的特性，希望在經過繁衍後能自動找出一個最佳的參數組合，有效的降低所需的人力及時間。

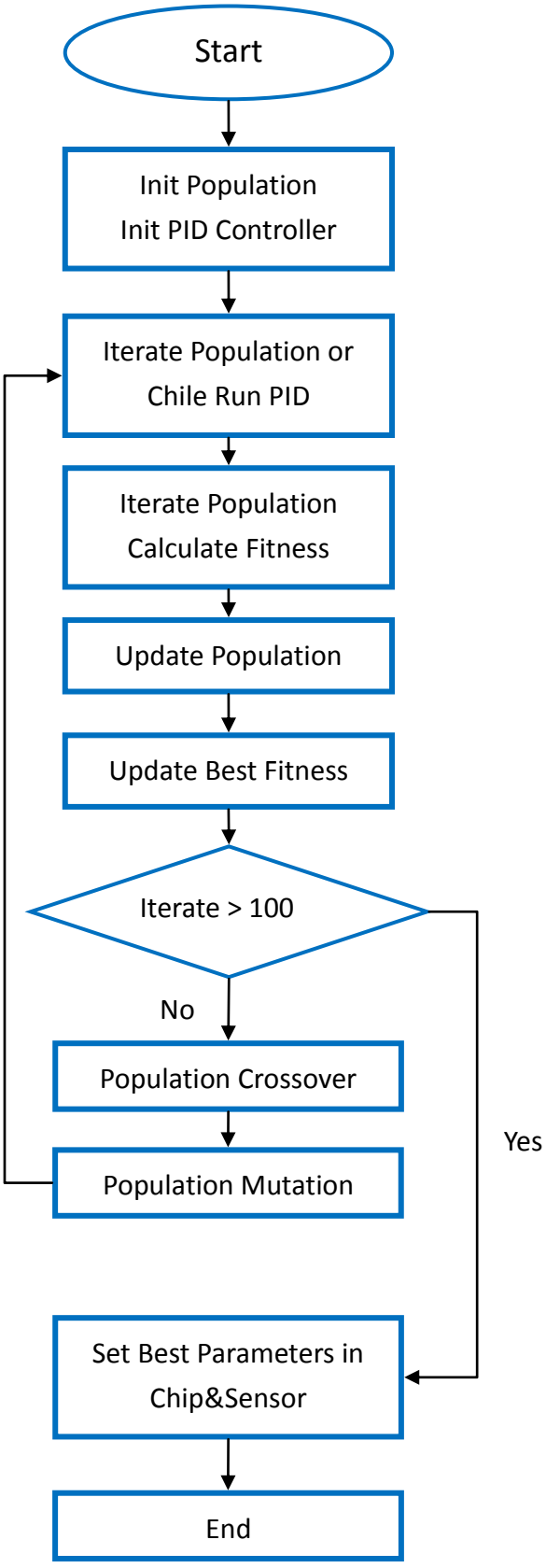
目前所參考的文獻資料中有部分提到使用轉移函數可以模擬出實驗的電路來測試。但轉移函數會根據不同的電路裝置而不同，實驗用的轉移函數需要再花時間設計。而實體電路可以另外取得的情況下並不會造成太大的影響。另外參考的文獻的基因演算法部分，在複製階段選擇速度較輪盤式選擇法快的競爭式選擇法。考量到實驗過程不會花太多時間以及輪盤式選擇法有著不會限於區域最佳解的優點，最後決定使用輪盤式選擇法。

三、 提出的方案

(一)預期目標詳述

一般在家電的控制上若想要達到省電除了改善實體元件外，也可以藉由調整最適輸出的平均電壓準位，這通常利用 PWM（Pulse Width Modulation）來控制，但是若想要自動的且剛好調整 PWM 則必須要由 PID（Proportional-Integral-Derivative）演算法來調整，PID 等同於一個濾波器（Filter）再調整增益參數時由人工調整通常需要非常久的時間才能調整出符合的參數，因此為了節省效率我們希望利用基因演算法來自動的找出最佳的參數，來節省人力的成本與產品的可用性，為此我們示範利用一個發光二極體（LED）與光敏電阻（CDs）互相配合來因應環境光（Ambient Light）的強弱來自動調整亮度。

(二)系統流程圖

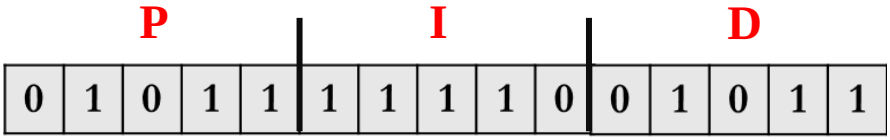

Start
Init Population
Init PID Controller
Iterate Population or
Chile Run PID
Iterate Population
Calculate Fitness
Update Population
Update Best Fitness
Iterate > 100
No
Yes
Population Crossover
Population Mutation
Set Best Parameters in
Chip&Sensor
End

(三)基因演算法的方案詳述

在利用基因演算法之前，必須先針對編碼、適應函數及初始族群產生方式等方法先做定義，分別敘述如下：

A. 基因編碼

由於本專題的目的是求出 PID 控制器的三個參數值，因此我們直接將參數 Kp、Ki 和 Kd 串接，用以當作基因的編碼。



如上圖所示，基因長度為 15bit，三個參數各用 5bit 來代表其數值範圍 0~31。

B. 產生初始族群

初始族群是使用亂數編碼的方式產生，並且我們將每一代母體的數量限制為 30。

C. 適應函數

為了求出 PID 控制器的三個參數最佳值，本專題採用兩個判斷指標來作為基因演化過程中生存或淘汰的依據：

1. 累計誤差值(絕對誤差積分)

藉由基因中包含的一組參數，計算出其調整輸出值直到符合本系統的設定值(Setpoint)為止，所累計的誤差值。此數值愈小代表基因的適應值愈高。

2. 誤差收斂時間

基因中包含的 PID 參數若能使系統在最短時間內達到設定值，即代表誤差收斂時間短。此數值愈小代表基因的適應值愈高。

但由於本系統的誤差值需要藉助感測器回傳數值計算，因此為了控

制花費的時間，我們讓每一組 PID 參數皆可調整輸入值 50 次，最後再藉由累計誤差值的倒數來作為適應值的評估。

D. 複製

採用輪盤式選擇法，適應函數值愈高的基因會有較高的機率被複製到交配池進行交配，可以使群體中基因的適應值不斷的接近最佳解，本專題設定的交配池大小為 10。

E. 交配運算

在選出的兩個基因中，隨機選取兩個交配點作交配，交換此兩個點的位元。交配率設為 0.5，會在 0.5 附近作測試調整。

F. 突變運算

挑選出隨機一個基因的一個位元來作突變，採用單點突變。突變率設為 0.1，會在 0.5 附近作測試調整。

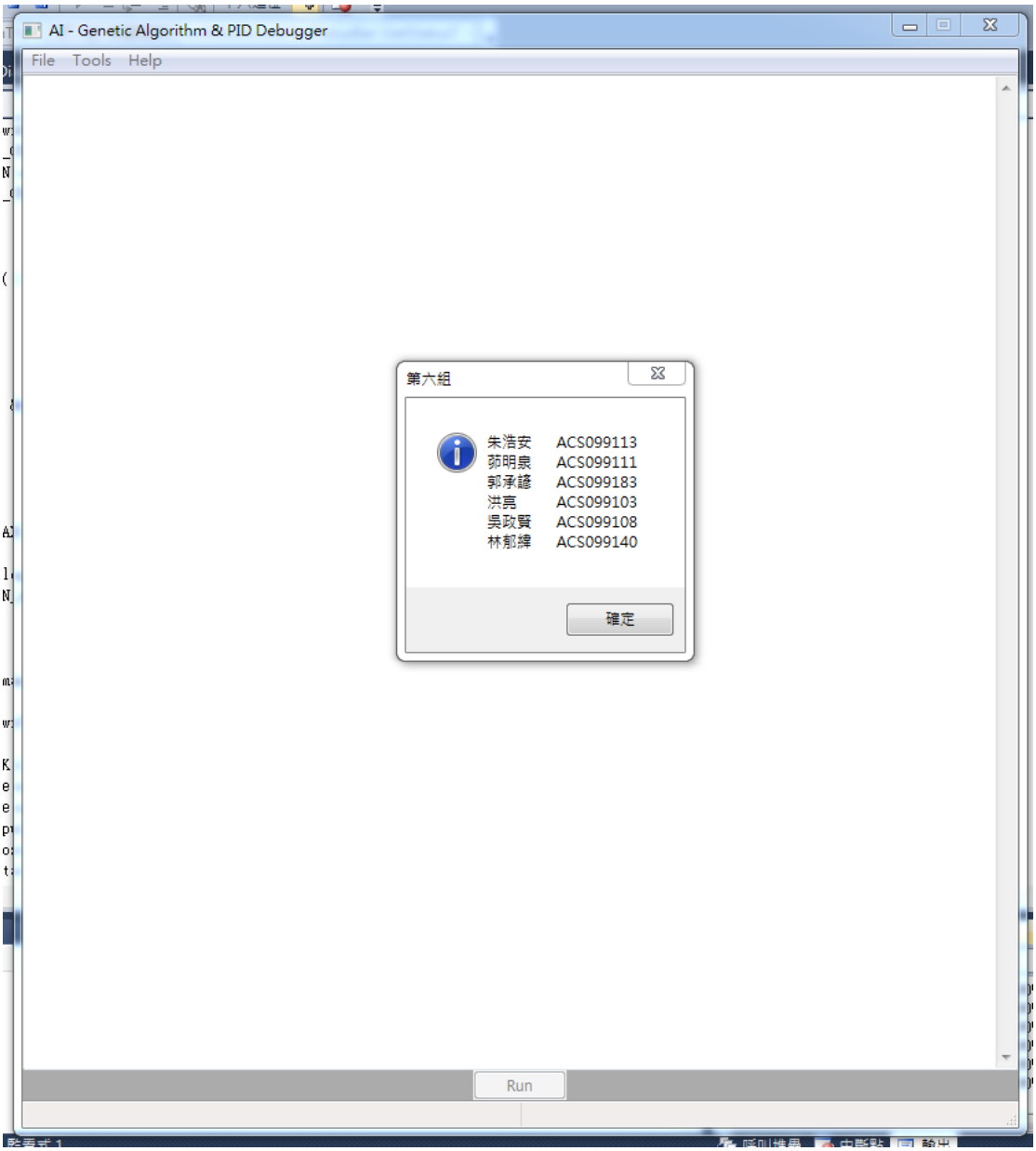
G. 結束條件

在基因演化 100 代後，基因演算法便會停止，目前所得的最佳適應函數值的基因即為最佳解。

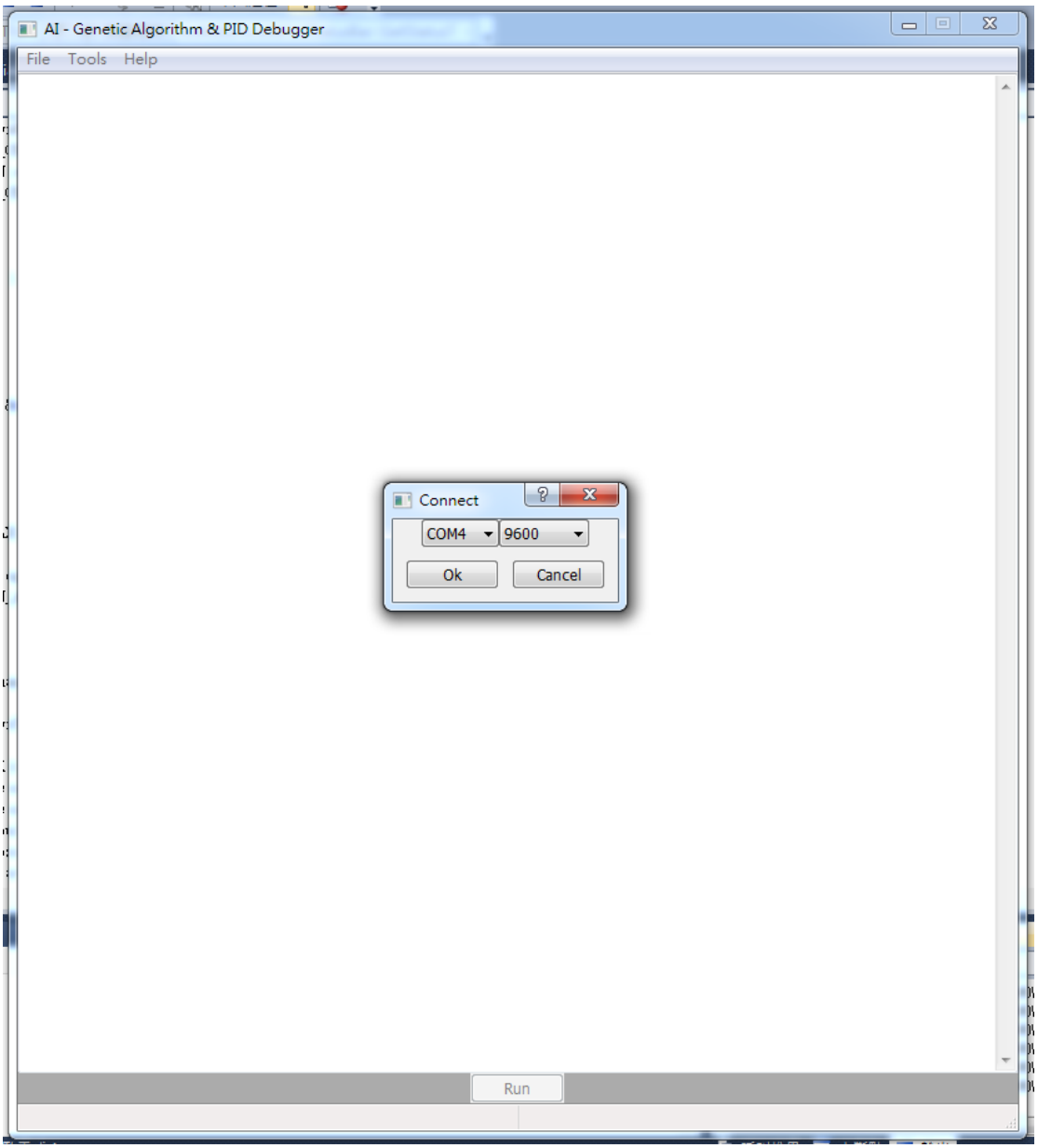
四、 實驗過程與結果

(一) PC 端控制程式

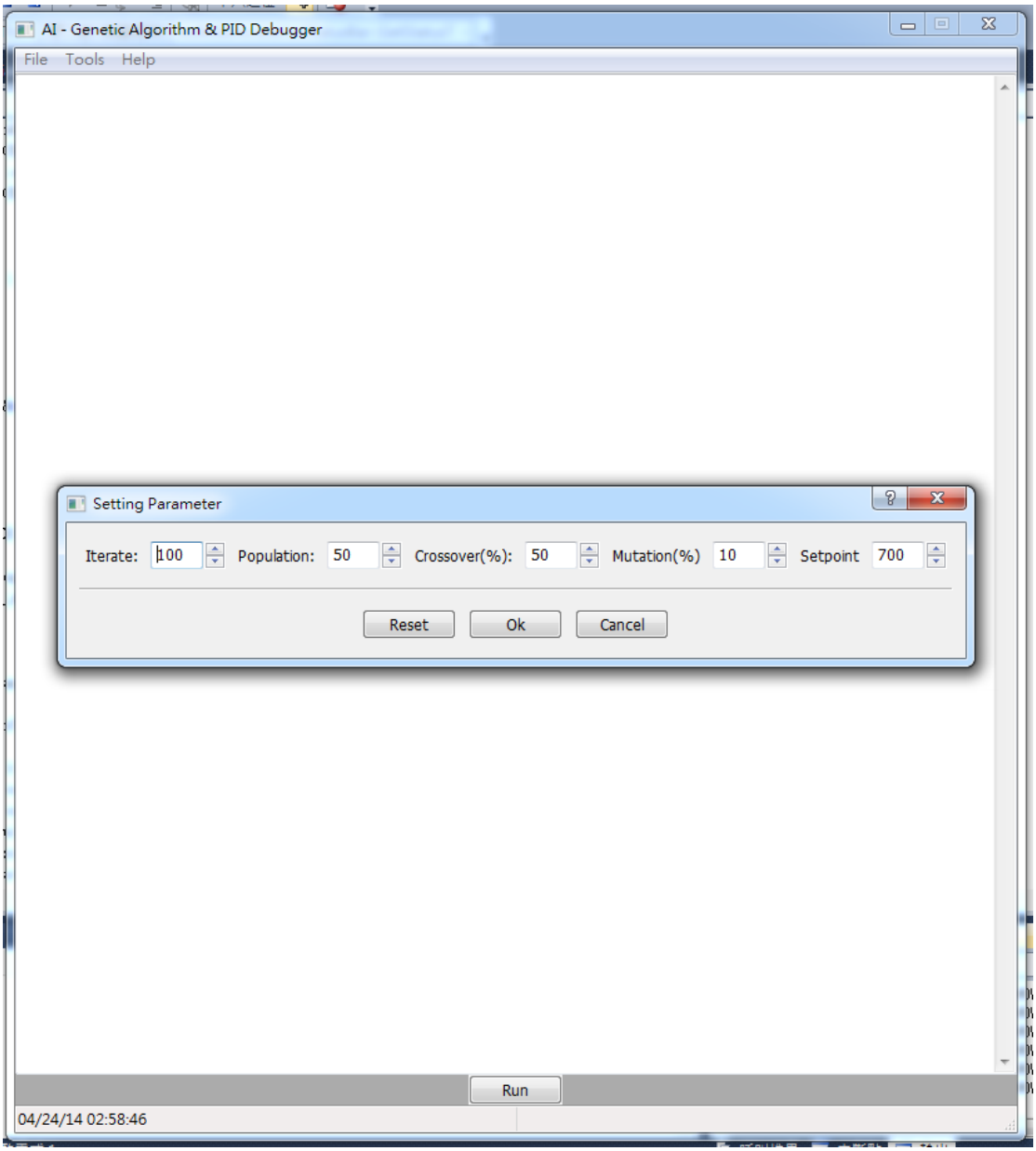
A. 主程式介面


AI - Genetic Algorithm & PID Debugger
File Tools Help
第六組
朱浩安 ACS099113
茆明泉 ACS099111
郭承諺 ACS099183
洪宮 ACS099103
吳政賢 ACS099108
林郁婷 ACS099140
確定
Run

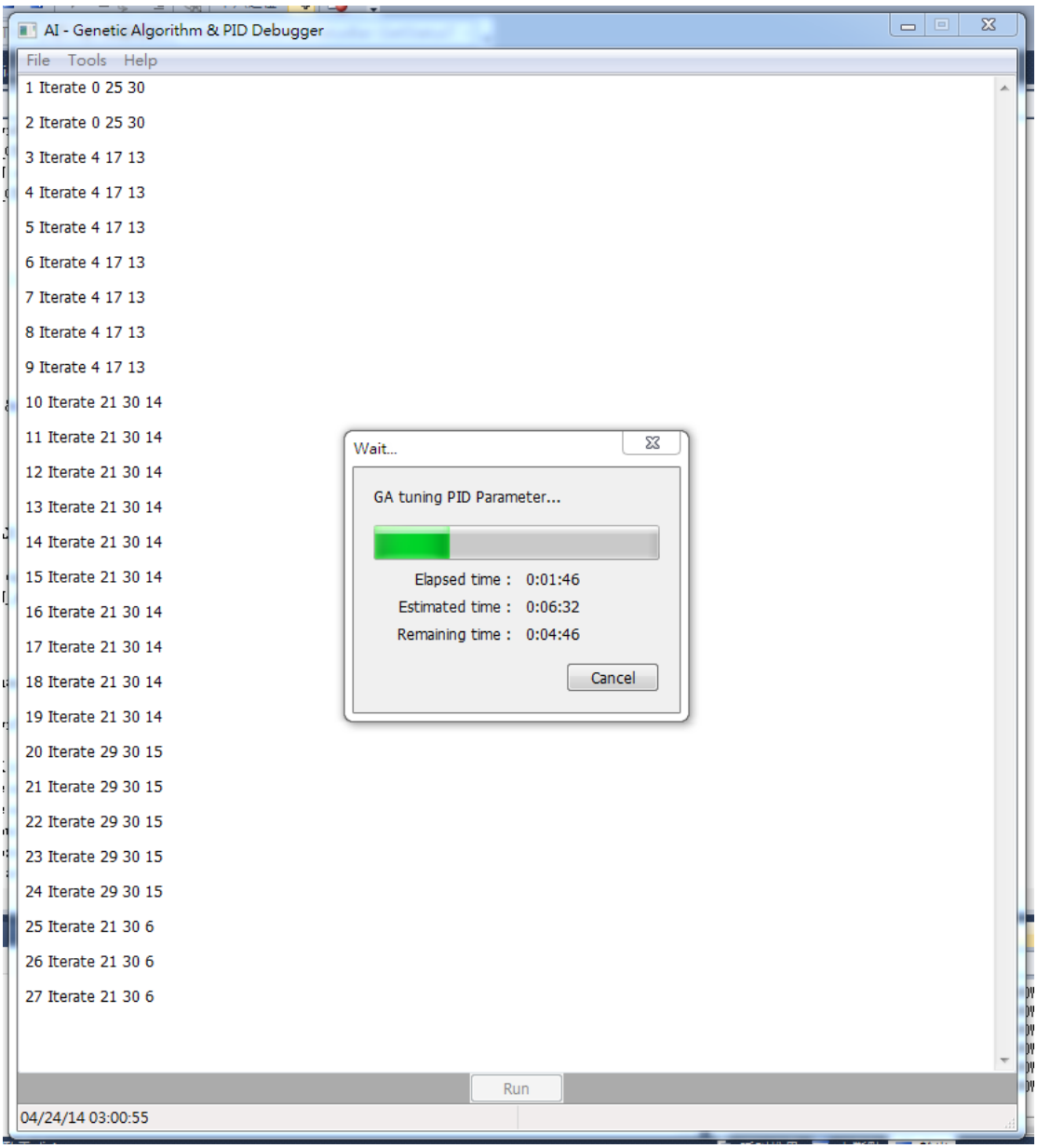
B. 連接控制板設定


AI - Genetic Algorithm & PID Debugger
File
Tools
Help
Connect
COM4
9600
Ok
Cancel
Run

C. 基因演算法參數設定



D. 執行並取得控制板回傳結果



(二)控制板主要程式碼

A. PID

```
void PID::Computing()
{
   if(on_off == OFF){
      return ;
   }

   unsigned long now = micros();

   if((now - last_time) >= sample_time){
       count++;
      double error = (*setpoint - *input);

      integral += (Ki * error);
      if(integral > max){
     integral = max;
      }
      else if(integral < min){
     integral = min;
      }

      double diff_input = (*input - last_input);

      *output = (Kp * error) + integral - (Kd * diff_input);

      if(*output > max){
     *output = max;
      }
      else if(*output < min){
     *output = min;
      }

      last_input = *input;
      last_time = now;
   }
}
```

```
void PID::SetTunings(double _Kp,double _Ki,double _Kd)
{
   if((_Kp < 0) || (_Ki < 0) || (_Kd < 0)){
      return ;
   }

   double SampleTimeInSec = ((double)sample_time) / 1000000.0f;

   Kp = _Kp;
   Ki = _Ki * SampleTimeInSec;
   Kd = _Kd / SampleTimeInSec;

   if(direction != REVERSE){
      Kp = (0 - Kp);
      Ki = (0 - Ki);
      Kd = (0 - Kd);
   }
}
```

B. GA

```
typedef struct
{
    int DNA:GENE_LENGTH;        //基因長度15bit
    int select: 1;              //判斷此基因是否被選擇1bit
    int pid_run: 1;             //判斷此基因是否跑過PID 1bit
    unsigned long error_total;  //總累計誤差
}Gene;
```

```
void SwapBit(Gene *a, Gene *b, int bit_no1, int bit_no2){      //交換a,b的第bit_no個位元
    int i, tempa=a->DNA, tempb=b->DNA;
    tempa >>= GENE_LENGTH-bit_no1;
    tempb >>= GENE_LENGTH-bit_no2;
    if((tempa&=0x1) != (tempb&0x1)){
        int temp1=1, temp2=1;
        for(i=0; i<GENE_LENGTH-bit_no1; i++){temp1*=2;}
        for(i=0; i<GENE_LENGTH-bit_no2; i++){temp2*=2;}
        if(tempa==0){
            a->DNA+=temp1; b->DNA-=temp2;
        }else{
            a->DNA-=temp1; b->DNA+=temp2;
        }
    }
}
```

```
void Mutation()                                          //突變：單點突變
{
    int counter = 0;
    int mutation_bits_num = MAX * GENE_LENGTH * MUTATION_RATE;//突變bit數
    int random_gene, random_bit;                         //隨機基因、位元

    while(counter < mutation_bits_num){                  //突變位元
        random_gene = rand(MAX) - 1;
        random_bit = rand(GENE_LENGTH) - 1;

        int i, temp = population[random_gene].DNA;
        temp >>= GENE_LENGTH - random_bit;

        temp &= 0x1;
        int factor = 1;
        for(i=0; i<GENE_LENGTH-random_bit; i++){factor*=2;}
        if(temp==0){
            population[random_gene].DNA+=factor;
        }else{
            population[random_gene].DNA-=factor;
        }

        counter++;
    }

    memcpy(&population[0], &best_gene, sizeof(Gene)); //保留最佳母體
}

void FitnessFunc()              //計算Fitness，愈大愈好
{
    int max = 0;                //目前Fitness最大值
    int best_gene_position = 0; //最好的基因位置
    int error = 0;              //PID回傳結果
    int Magnification = 100;    //放大倍率

    for(int i = 0;i < MAX;i++) {
       error = _PID(&population[i]);
       if(error != 0)           //取誤差值倒數
            population[i].error_total = (unsigned long)Magnification / error;
       if(max < population[i].error_total) {
            max = population[i].error_total;
            best_gene_position = i;
       }
    }
    best_gene = population[best_gene_position]; //選出最好基因
}
```

```
unsigned long _PID(Gene *gene)
{
  unsigned long error_total = 0;
  analogWrite(3,255);
  pid.SetTunings(gene->DNA & 0x01F,
  (gene->DNA >> 5) & 0x01F,
  (gene->DNA >> 10) & 0x01F);
  //pid.SetOnOff(ON);
  for(int i = 0;i < ((22 * (gene->DNA >> 10)) + 300);++i){
    input = analogRead(A0) / 4;
    pid.Computing();
    analogWrite(3,output);
    error_total += abs(input - setpoint);
  }
  //pid.SetOnOff(OFF);
  error_total /= ((22 * (gene->DNA >> 10)) + 300);
  return error_total;
}
```

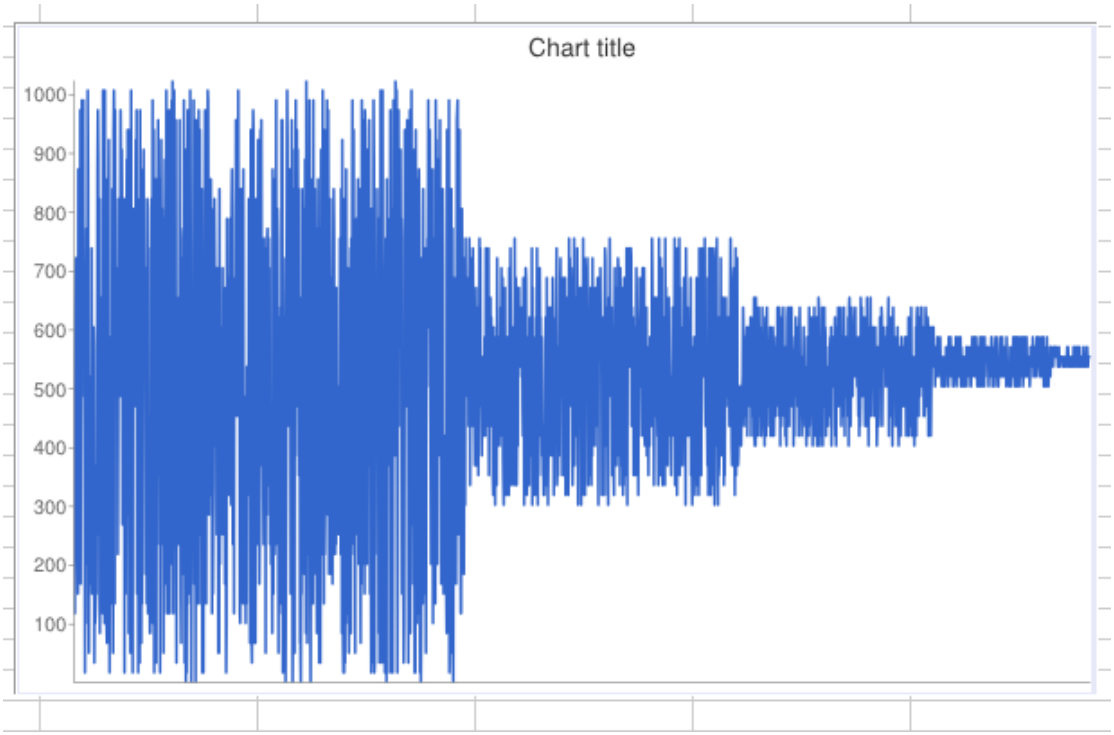
C. MCU

```
void Paser()
{
  while(Serial.available()){
    flag = true;
    parameter[count++] = Serial.read();
  }
  if(flag && (count == PMAX)){
    flag = false;
    count = 0;
    switch (parameter[0])
    {
      case 'r':
      {
        unsigned char chk = parameter[PMAX-1];
        CheckSum();
        if(chk == parameter[PMAX-1]){
          unsigned int temp = 0;
          temp += (parameter[1] * 255);
          temp += (parameter[2]);
          ITERA = temp;
          MAX = parameter[3];
          CROSSOVER_RATE = parameter[4] * 0.001f;
          MUTATION_RATE = parameter[5] * 0.001f;
          temp = 0;
          temp += (parameter[6] * 255);
          temp += (parameter[7]);
          setpoint = temp/4;
          ga_run = true;

        }
        else{
          Serial.println("CheckSum Error");
        }
        break;
      }
      case 0x0:
      {
        unsigned char chk = parameter[PMAX-1];
        CheckSum();
        if(chk == parameter[PMAX-1]){
          for(int i = 1;i < PMAX-1;++i){
            if(parameter[i] == i){

              ga_run ^= 0x1;
            }
          }
        }
        break;
      }
      default:
```

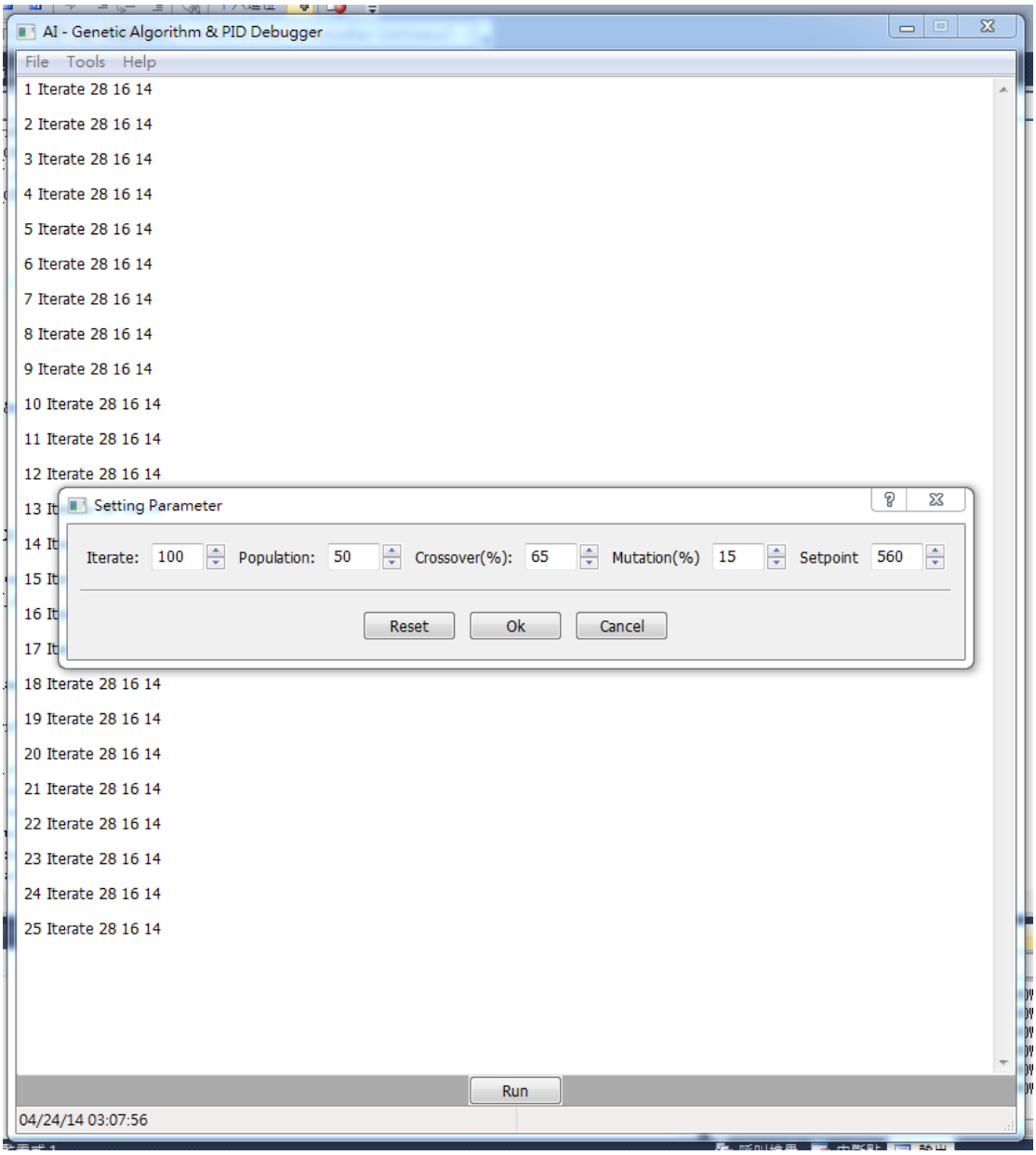
```
while(ga_run){
  Paser();
  int i = 1;
  InitGene();
  FitnessFunc();
  USART1_Send_String(i);
  USART1_Send_String(" Iterate ");
  USART1_Send_String(best_gene.DNA & 0x01F);
  USART1_Send_String(" ");
  USART1_Send_String((best_gene.DNA >> 5) & 0x01F);
  USART1_Send_String(" ");
  USART1_Send_String((best_gene.DNA >> 10) & 0x01F);
  for(i = 2;(i <= ITERA) && ga_run;++i){
    Paser();
    Crossover();
    Mutation();
    FitnessFunc();
    USART1_Send_String(i);
    USART1_Send_String(" Iterate ");
    USART1_Send_String(best_gene.DNA & 0x01F);
    USART1_Send_String(" ");
    USART1_Send_String((best_gene.DNA >> 5) & 0x01F);
    USART1_Send_String(" ");
    USART1_Send_String((best_gene.DNA >> 10) & 0x01F);
  }
  ga_run = false;
}
```

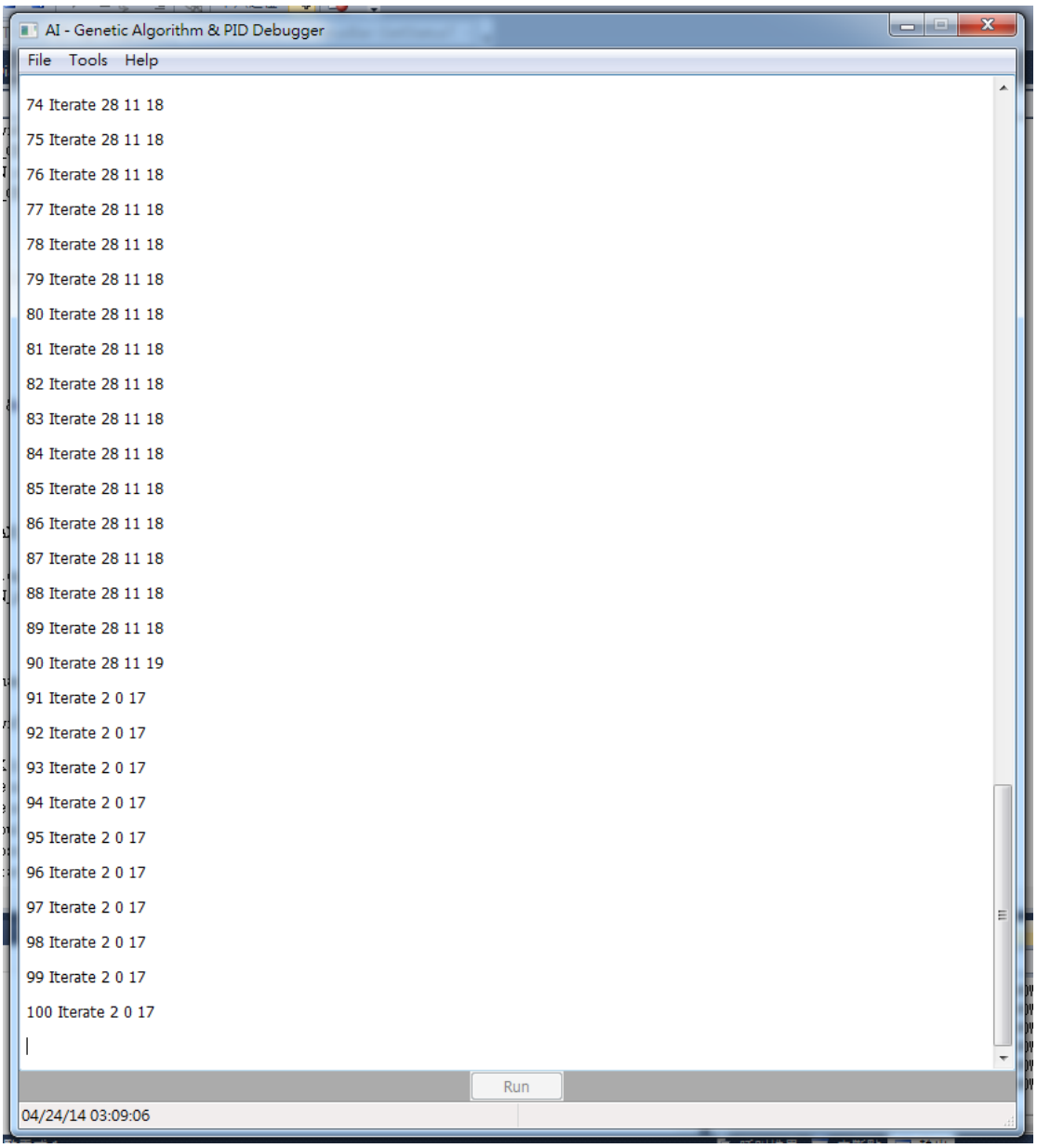
(三)實驗結果

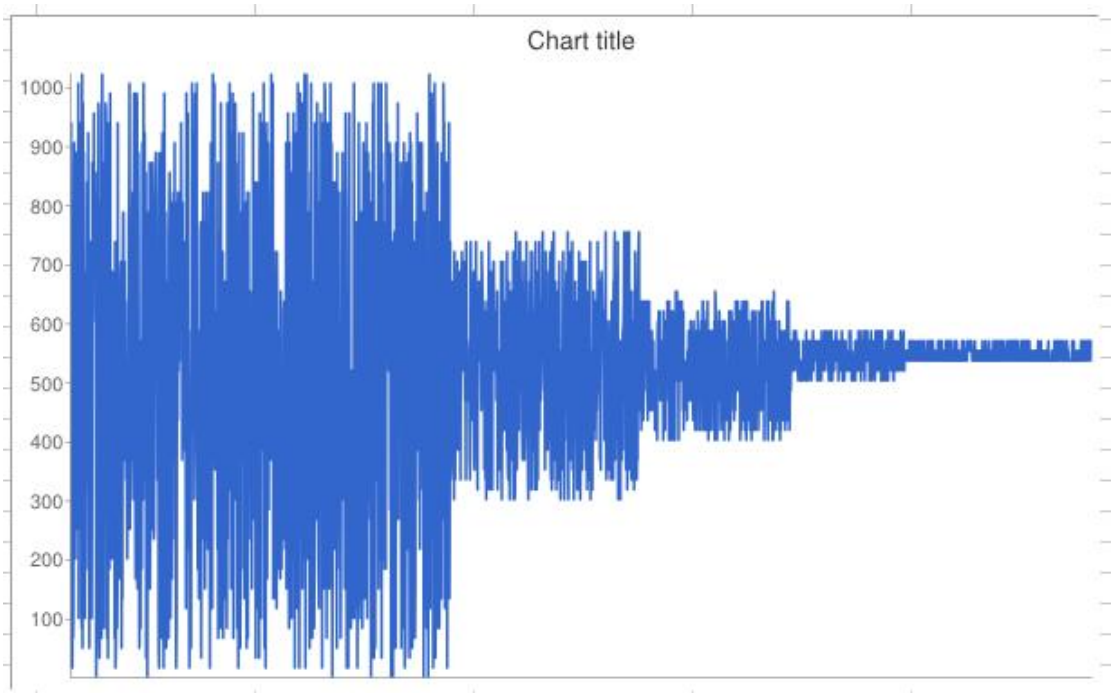
A. 利用 Ziegler–Nichols method 調整之 PID 運行結果（Kp = 1.7 Ki = 1.15 Kd = 0.5）



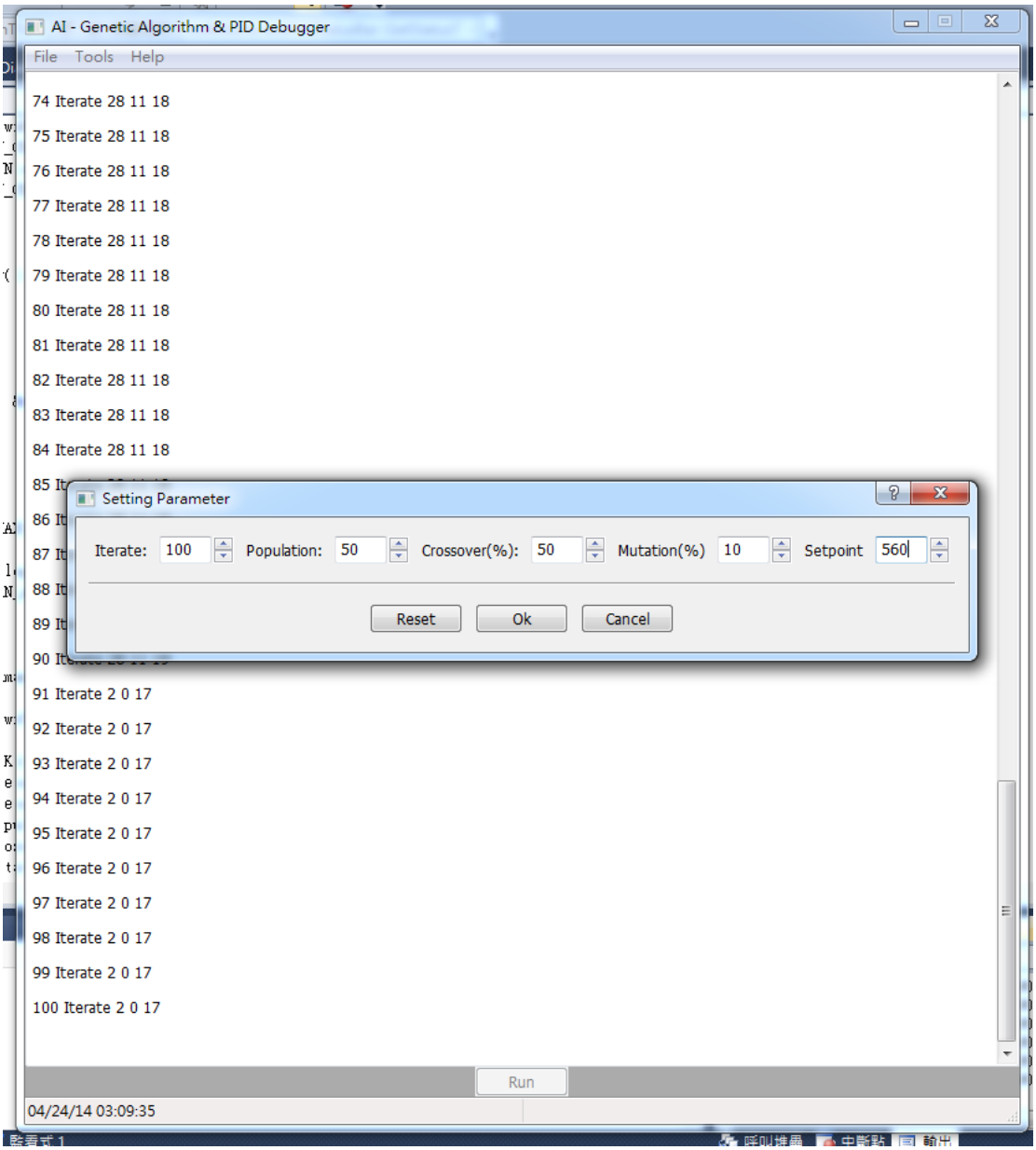
B.　利用基因演算法調整結果（一）（執行時間：00:06:53）

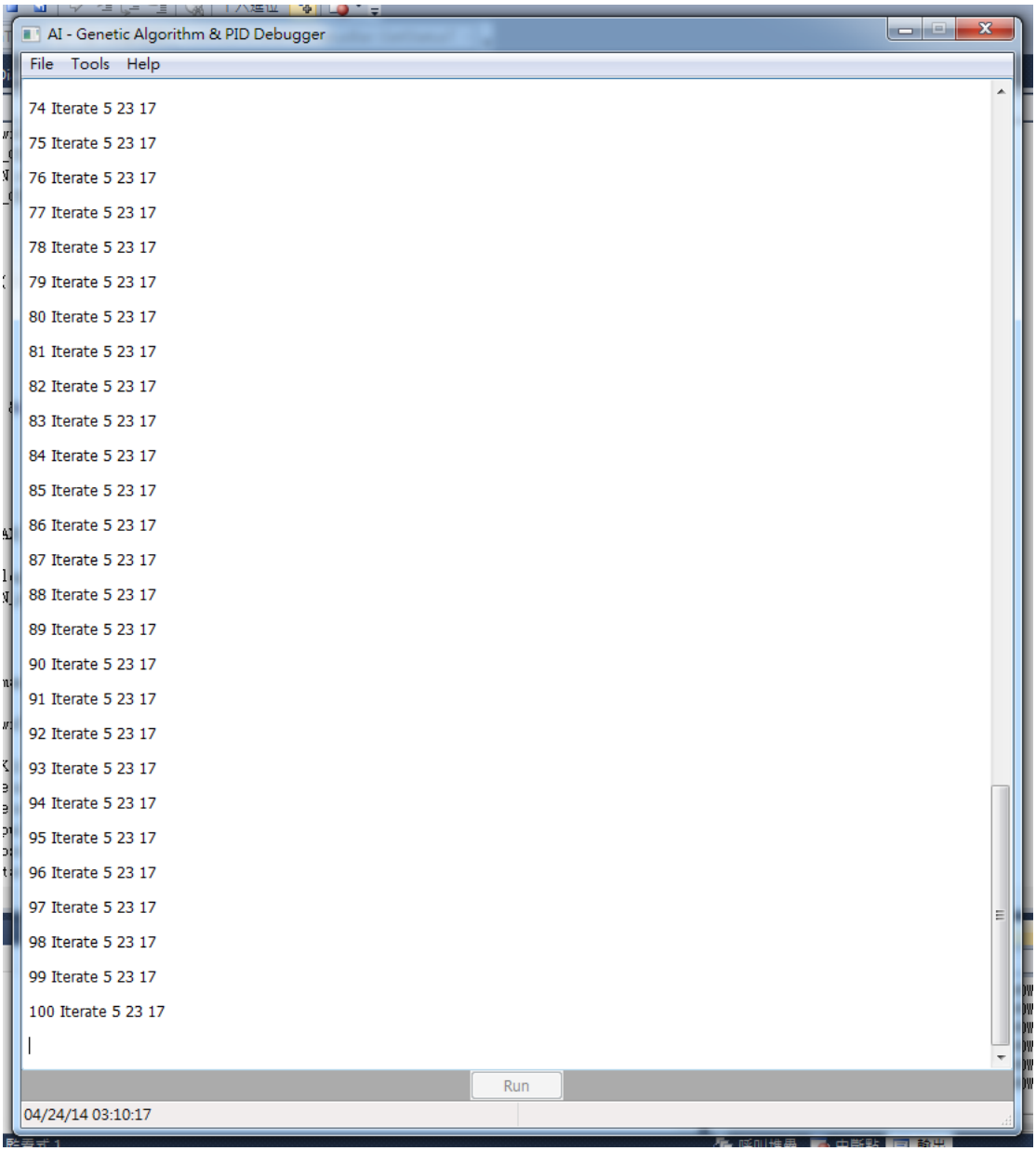

AI - Genetic Algorithm & PID Debugger
File Tools Help
1 Iterate 28 16 14
2 Iterate 28 16 14
3 Iterate 28 16 14
4 Iterate 28 16 14
5 Iterate 28 16 14
6 Iterate 28 16 14
7 Iterate 28 16 14
8 Iterate 28 16 14
9 Iterate 28 16 14
10 Iterate 28 16 14
11 Iterate 28 16 14
12 Iterate 28 16 14
Setting Parameter
Iterate: 100
Population: 50
Crossover(%): 65
Mutation(%) 15
Setpoint 560
Reset
Ok
Cancel
18 Iterate 28 16 14
19 Iterate 28 16 14
20 Iterate 28 16 14
21 Iterate 28 16 14
22 Iterate 28 16 14
23 Iterate 28 16 14
24 Iterate 28 16 14
25 Iterate 28 16 14
Run
04/24/14 03:07:56

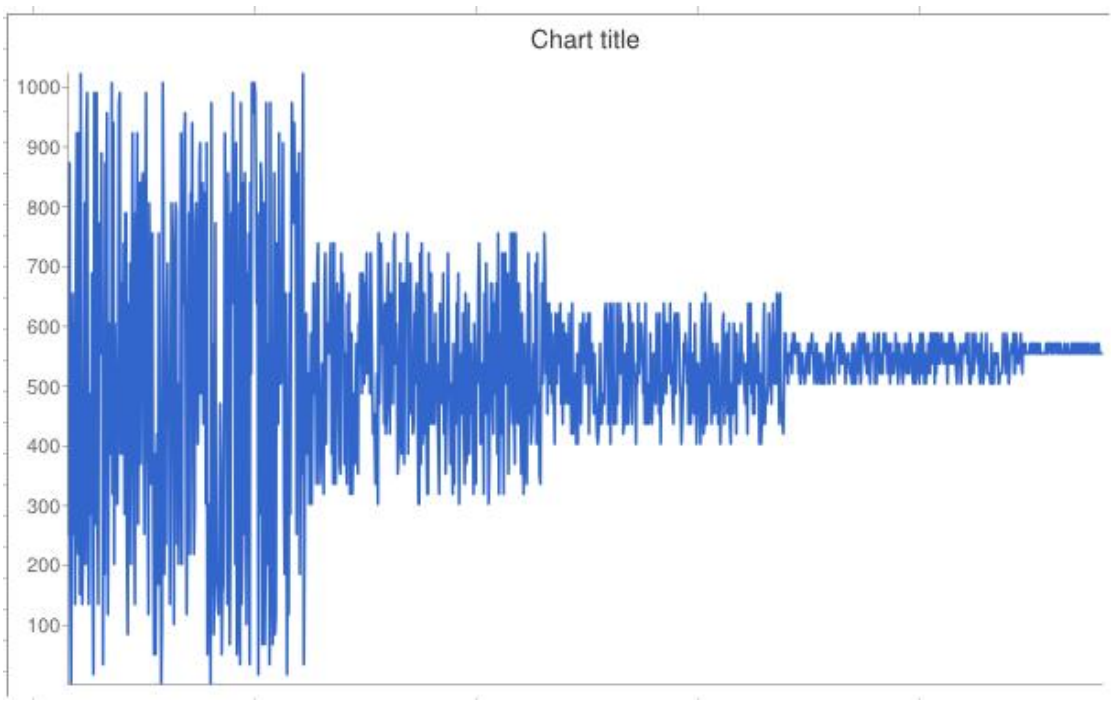

AI - Genetic Algorithm & PID Debugger
File Tools Help
74 Iterate 28 11 18
75 Iterate 28 11 18
76 Iterate 28 11 18
77 Iterate 28 11 18
78 Iterate 28 11 18
79 Iterate 28 11 18
80 Iterate 28 11 18
81 Iterate 28 11 18
82 Iterate 28 11 18
83 Iterate 28 11 18
84 Iterate 28 11 18
85 Iterate 28 11 18
86 Iterate 28 11 18
87 Iterate 28 11 18
88 Iterate 28 11 18
89 Iterate 28 11 18
90 Iterate 28 11 19
91 Iterate 2 0 17
92 Iterate 2 0 17
93 Iterate 2 0 17
94 Iterate 2 0 17
95 Iterate 2 0 17
96 Iterate 2 0 17
97 Iterate 2 0 17
98 Iterate 2 0 17
99 Iterate 2 0 17
100 Iterate 2 0 17
Run
04/24/14 03:09:06



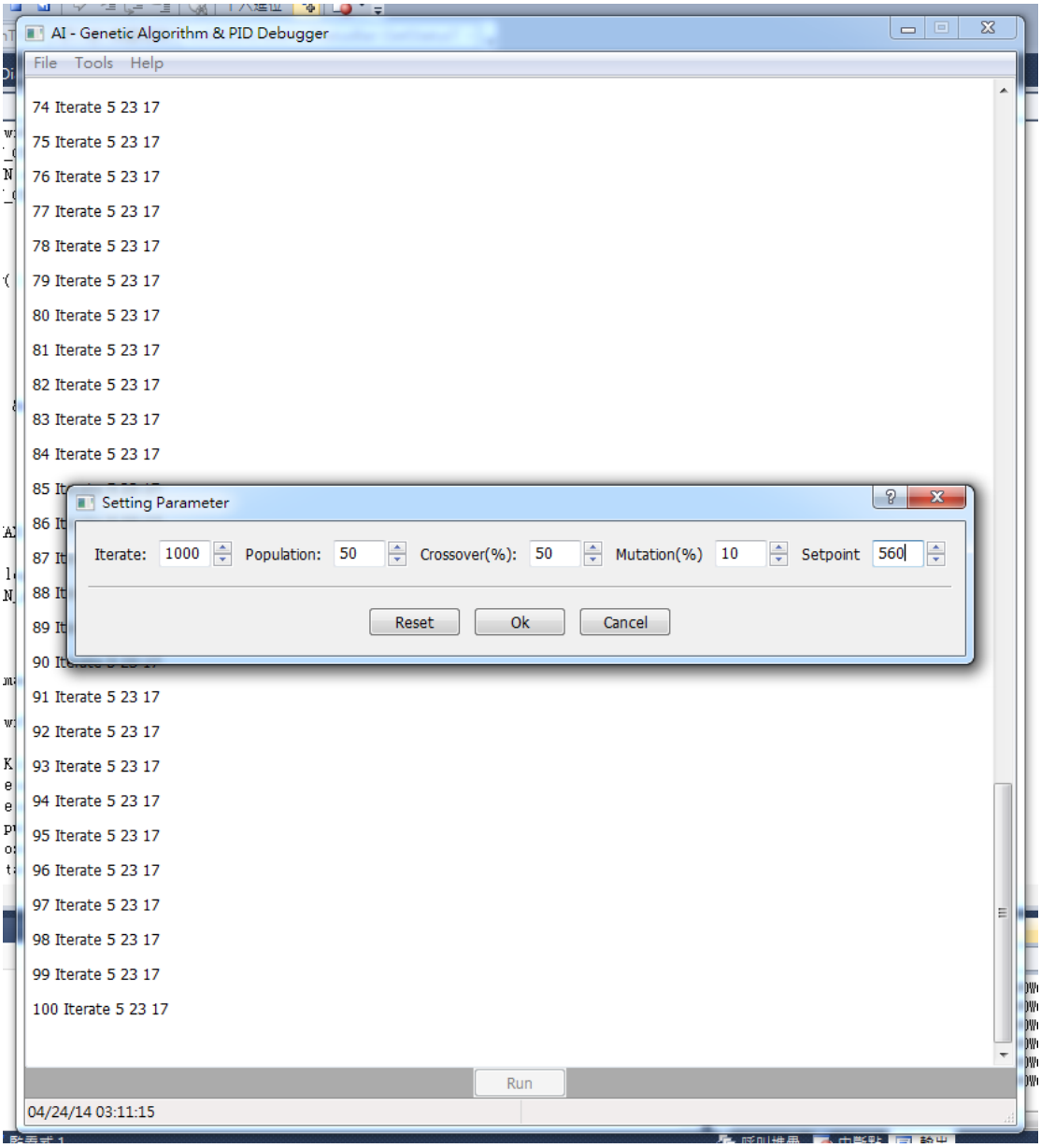
C. 利用基因演算法調整結果（二）（執行時間：00:04:53）

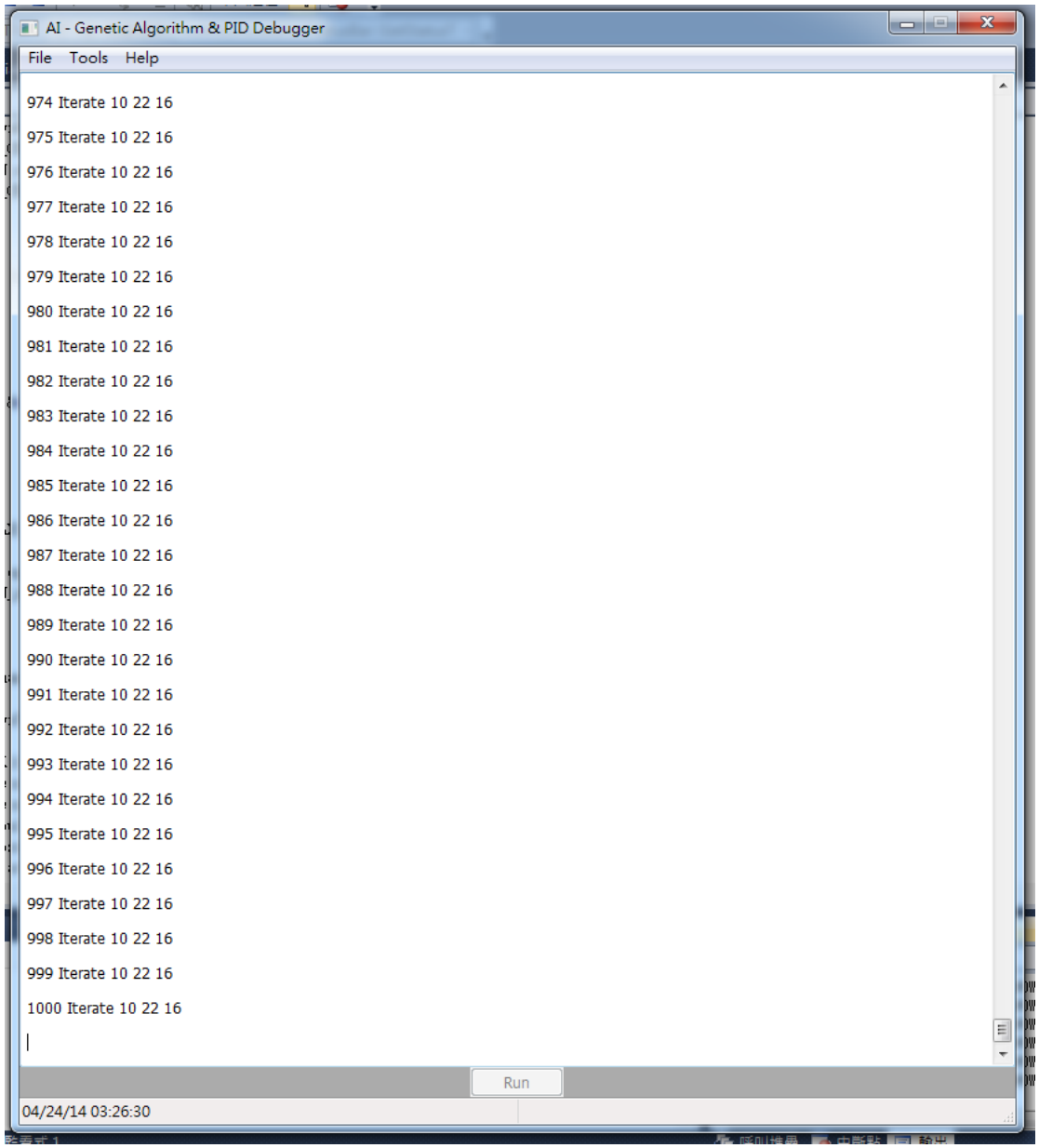

AI - Genetic Algorithm & PID Debugger
File Tools Help
74 Iterate 28 11 18
75 Iterate 28 11 18
76 Iterate 28 11 18
77 Iterate 28 11 18
78 Iterate 28 11 18
79 Iterate 28 11 18
80 Iterate 28 11 18
81 Iterate 28 11 18
82 Iterate 28 11 18
83 Iterate 28 11 18
84 Iterate 28 11 18
Setting Parameter
Iterate: 100 Population: 50 Crossover(%): 50 Mutation(%) 10 Setpoint 560
Reset Ok Cancel
91 Iterate 2 0 17
92 Iterate 2 0 17
93 Iterate 2 0 17
94 Iterate 2 0 17
95 Iterate 2 0 17
96 Iterate 2 0 17
97 Iterate 2 0 17
98 Iterate 2 0 17
99 Iterate 2 0 17
100 Iterate 2 0 17
Run
04/24/14 03:09:35

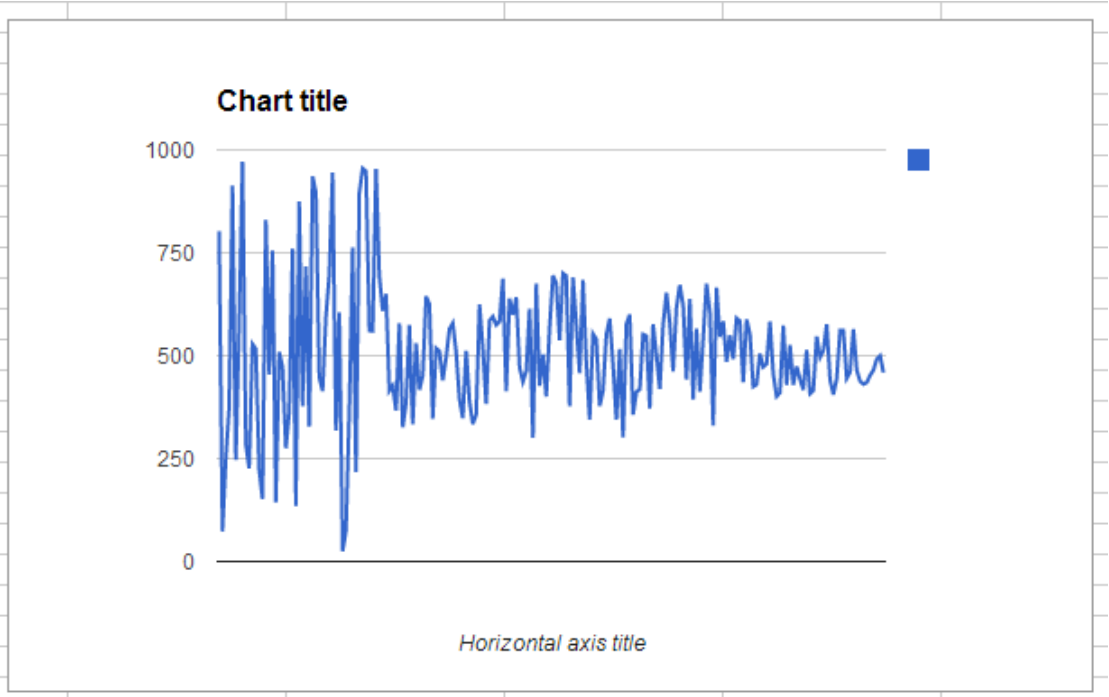

AI - Genetic Algorithm & PID Debugger
File Tools Help
74 Iterate 5 23 17
75 Iterate 5 23 17
76 Iterate 5 23 17
77 Iterate 5 23 17
78 Iterate 5 23 17
79 Iterate 5 23 17
80 Iterate 5 23 17
81 Iterate 5 23 17
82 Iterate 5 23 17
83 Iterate 5 23 17
84 Iterate 5 23 17
85 Iterate 5 23 17
86 Iterate 5 23 17
87 Iterate 5 23 17
88 Iterate 5 23 17
89 Iterate 5 23 17
90 Iterate 5 23 17
91 Iterate 5 23 17
92 Iterate 5 23 17
93 Iterate 5 23 17
94 Iterate 5 23 17
95 Iterate 5 23 17
96 Iterate 5 23 17
97 Iterate 5 23 17
98 Iterate 5 23 17
99 Iterate 5 23 17
100 Iterate 5 23 17
Run
04/24/14 03:10:17



D. 利用基因演算法調整結果（三）（執行時間：01:01:03）


AI - Genetic Algorithm & PID Debugger
File Tools Help
74 Iterate 5 23 17
75 Iterate 5 23 17
76 Iterate 5 23 17
77 Iterate 5 23 17
78 Iterate 5 23 17
79 Iterate 5 23 17
80 Iterate 5 23 17
81 Iterate 5 23 17
82 Iterate 5 23 17
83 Iterate 5 23 17
84 Iterate 5 23 17
Setting Parameter
Iterate: 1000 Population: 50 Crossover(%): 50 Mutation(%) 10 Setpoint 560
Reset Ok Cancel
91 Iterate 5 23 17
92 Iterate 5 23 17
93 Iterate 5 23 17
94 Iterate 5 23 17
95 Iterate 5 23 17
96 Iterate 5 23 17
97 Iterate 5 23 17
98 Iterate 5 23 17
99 Iterate 5 23 17
100 Iterate 5 23 17
Run
04/24/14 03:11:15


AI - Genetic Algorithm & PID Debugger
File Tools Help
974 Iterate 10 22 16
975 Iterate 10 22 16
976 Iterate 10 22 16
977 Iterate 10 22 16
978 Iterate 10 22 16
979 Iterate 10 22 16
980 Iterate 10 22 16
981 Iterate 10 22 16
982 Iterate 10 22 16
983 Iterate 10 22 16
984 Iterate 10 22 16
985 Iterate 10 22 16
986 Iterate 10 22 16
987 Iterate 10 22 16
988 Iterate 10 22 16
989 Iterate 10 22 16
990 Iterate 10 22 16
991 Iterate 10 22 16
992 Iterate 10 22 16
993 Iterate 10 22 16
994 Iterate 10 22 16
995 Iterate 10 22 16
996 Iterate 10 22 16
997 Iterate 10 22 16
998 Iterate 10 22 16
999 Iterate 10 22 16
1000 Iterate 10 22 16
Run
04/24/14 03:26:30



五、 討論與分析

在實作基因演算法來調整 PID 參數的過程中，常遇到許多不如預期的狀況。以下列出實作時所遇到問題的解決方法和說明。

- **最佳子代的適應值有時候會小於上一代的最佳母體**
- 解決方法：

  由於在演化過程中，擁有最佳適應值的母體有時候會因為交配或突變而降低適應值，導致最佳子代的適應值不如上一代，基因演算法必須演化更多代才能達到預期的結果，因此我們改變基因演算法的流程，將最佳母代直接複製到下一代的群體中，意味著最佳母代不會因為交配或突變而導致適應值降低的狀況，藉此來確保每一代的演化都會讓基因的適應值更接近目標。

- **基因演算法計算適應值的時間過長**
- 解決方法：

由於本專題計算適應值必須要藉由 PID 演算法，經由發光二極體及光敏電阻來得到每一組基因參數對系統的收斂情況，因此時間會比一般在電腦中的數值計算來的長，為了改善計算適應值的效率，我們在基因的資料結構增加了一個變數 pid_run，用以判斷此基因是否在交配或突變的過程發生改變，若否，則適應函數會跳過此基因的適應值計算，以節省不必要的計算時間。

- **突變率若設定為 0.2 以上時，**

  子代的適應值經常會變得沒有規律且變化過大

- 解決方法：

  由於突變率在 0.2 以上時會發生這種情形，因此本專題在測試時就將突變率設為 0.1，僅改變交配率來進行測試

六、 結論

為了有效的分析整個實驗的結果，我們記錄了每代所產生的結果，並將其繪製成圖表以利於分析，根據我們所記錄的圖表結果顯示，我們可以很明顯的看到隨著不斷的演化，最終我們的結果慢慢的趨於穩定的狀態，也就是說我們達成了我們所追求的目標－自動找出最適合的 PID 參數，也證明了基因演算法確實在 PID 參數的調整上，確實能夠幫助我們達到快速找到最佳解，使得誤差能迅速消除，並且減輕人工調整的負擔。

七、 參考資料

[1] 陳宏謀,“遺傳基因演算法求取 PID 控制器之參數的最佳解並與 Wang 法相比較”, 2005

[2] Kuo & F. Golnaraghi, “Automatic Control Systems_B. C.”

[3] 陳威韶, “To Apply Genetic Algorithms for Servomotor PID Controller Parameter Modulation”, 2004

[4] 陳俊宏 林信甫 謝谷典, “The Study of PID Controller Tuning by Using Genetic Algorithm”, 2001

八、 附錄：其他組解法

(一)暗棋

Gene：

[棋子編號 1][將作用之座標 1][棋子編號 2][將作用之座標 2][棋子編號 3][將作用之座標 3]

棋子編號:將要移動的棋子或要進行翻棋

將作用之座標:棋子移動後的位置或翻棋的位置

Fitness：

依照敵我方剩下的棋子種類與數量來決定棋子的分數

(對方可以吃掉我方棋子的數量/我方可以吃掉對方棋子的數量)，分數較高者較好。

(我方獲得分數 1*3-對方獲得分數 2*2+我方獲得分數 3)/對方獲得分數 2。

我方獲得分數 1:棋子編號 1 與將作用之座標 1 所吃掉對方棋子的分數

對方獲得分數 2:棋子編號 2 與將作用之座標 2 所吃掉我方棋子的分數

我方獲得分數 3:棋子編號 3 與將作用之座標 3 所吃掉對方棋子的分數

最後得出的數值越高越好。

Crossover

將選出基因的[[棋子編號][將作用之座標]]其中一組進行交換

Mutation

隨意改變[棋子編號]或[將作用之座標]其中一組

(二)數獨

基因編碼：

以題目的空格數做為基因編碼的長度，例如下圖的基因編碼長度為32，每格填入1到9其中一個數字。

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

適應函數：

以數獨的三個規則來設計適應函數，每行、每大格中1到9每個數字都只能出現一遍，若有符合時，則增加適應分數。

各規則的分數如下：

該大格內1到9均有：10000

該行內1到9均有：100

該列內1到9均有：100

由於每大格內包含1到9所有數字是一個相當重要的指標，所以將其權重設為1000，當任一基因的分數達到91800時即為最佳解。

適應函數的虛擬碼如下：

```
Fitness(gene A)     {
    Int   weight    =    0;
    Bool check [9] =    {false};
    Int   board[9][9]   =    將基因填入題目空格;

    For(int i=0; i<9; i++)  {
        For(int j=0; j<9;j++)   {
            Check[board[i][j]] =    true;
        }
        若 check 陣列均為 true，則 weight +=100;
        Check 陣列設回全 false;

      For(int j=0; j<9;j++){
```

```
                    Check[board[j][i]] =    true;
        }
        若 check 陣列均為 true，則 weight +=100;
        Check 陣列設回全 false;
    }

    //大格規則
    Int     offset_i   =    0, offset_j=    0;
    While(offset_i < 3)    {
      For(int i = offset_i*3; i < offset_i*3+3; i++)     {
          For(int j = offset_j*3; j < offset_j*3+3; j++)     {
              Check[board[i][j]] =    true;
            }
        }
        若 check 陣列均為 true 則 weight += 10000;
        Check 陣列初始為 false;
        Offset_j ++;
        If(offset_j == 3) {
          Offset_i++;;
          Offset_j = 0;
        }
    }
    Return weight;
}
```

Ex；參照上方數獨取 geneA= {3, 1, 2, 4, 7, 8, 6, 1, 2, 3, 1, 2, 3, 4, 5, 2, 4, 3, 2, 4, 8, 9, 3, 4, 2, 1, 8, 7, 7, 8, 2, 8}
根據適應函數所得出的結果為 10100

交配函數：
隨機取 geneA 和 geneB 中位於某一大格內的部分進行交換。
虛擬碼如下：

```
Crossover(geneA, geneB)    {
    任意取一大格所包含的空格區間;
    交換 geneAB 內該控格區間的數字;
}
```

Ex：取 geneA= {3, 1, 2, 4, 7, 8, 6, 1, 2, 3, 1, 2, 3, 4, 5, 2, 4, 3, 2, 4, 8, 9,

3, 4, 2, 1, 8, 7, 7, 8, 2, 8}

geneB = {4, 7, 8, 6, 7, 2, 6, 1, 2, 3, 1, 2, 3, 4, 5, 2, 4, 3, 2, 4, 8, 9, 3, 4, 2, 1, 8, 7, 7, 8, 2, 8}

若選中上方的大格進行交換，則產生的新的子代

ChildA = {3, 1, 2, 6, 7, 2, 6, 1, 2, 3, 1, 2, 3, 4, 5, 2, 4, 3, 2, 4, 8, 9, 3, 4, 2, 1, 8, 7, 7, 8, 2, 8}

ChildB = {4, 7, 8, 4, 7, 8, 6, 1, 2, 3, 1, 2, 3, 4, 5, 2, 4, 3, 2, 4, 8, 9, 3, 4, 2, 1, 8, 7, 7, 8, 2, 8}

突變函數：

隨機取 gene 內某一大格的空格重新隨機填入數字。

虛擬碼如下：

```
Mutation(geneA)  {
    任意取 gene 內任一大格的部分亂數重新填入;
}
```

Ex；取 geneA= {3, 1, 2, 4, 7, 8, 6, 1, 2, 3, 1, 2, 3, 4, 5, 2, 4, 3, 2, 4, 8, 9, 3, 4, 2, 1, 8, 7, 7, 8, 2, 8}

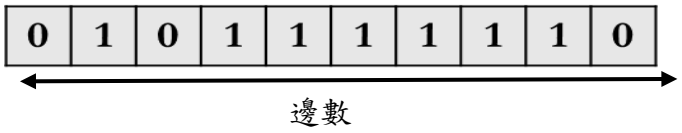
若隨機取右上方大格進行突變，則亂數重新填入的結果可能為

mutationGene = {3, 1, 2, 4, 7, 8, 6, 6, 5, 1, 1, 2, 3, 4, 5, 2, 4, 3, 2, 4, 8, 9, 3, 4, 2, 1, 8, 7, 7, 8, 2, 8}

(三)最短路徑

基因編碼

基因編碼使用資料結構為 Route 的陣列表示，產生初代母體時隨機生成 Route.enable，即代表是否要經過此路徑，如下圖所示。



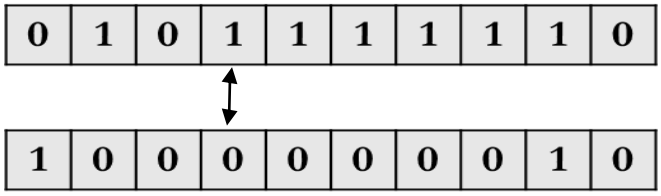

```
GENETIC_LENGTH =  邊總數量;
struct Route
{
    int enable; //是否經過
    int start_node, end_node;
    int distance:
};

struct Gene
{
    Route genes[GENETIC_LENGTH];
    int fitness;
};
```

適應函數

```
void fitness_calculate(Gene input)
{
    int fitness = 0;
    if(恰一個邊與起點相連){ fitness+=50; }
    if(恰一個邊與終點相連){ fitness+=50; }
    if(邊形成的路徑相連接){ fitness+=50; }
    fitness+=1000/路徑加總 distance;

    Gene.fitness = fitness;
}
```

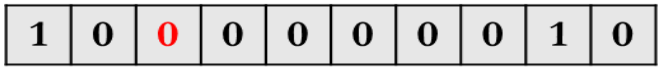
交配

將 genes[i].enable 做隨機單點交換



突變

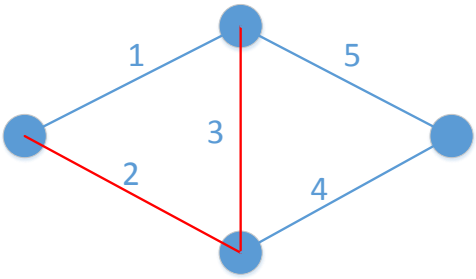
將 genes[i].enable 做隨機單點突變



舉例

假設路徑長度照編號排序分別為[2, 10, 20, 20, 4]，一組基因為

化為圖示即表示成



計算 fitness 值即為：

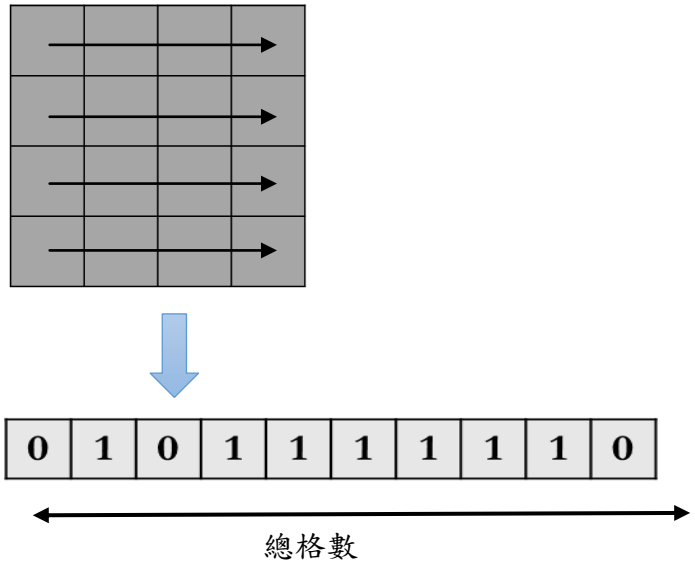
50(恰一個邊與起點相連)+50+(邊形成的路徑相連接)+1000/30(路徑總長倒數)

=133

(四)自走箱

基因編碼

基因編碼會由左到右將題目的地圖轉為一維陣列表示，產生初代母體群時隨機生成 0 或 1 即代表是否要經過此格，如下圖所示。



GENETIC_LENGTH = 總格數;
const bool map[GENETIC_LENGTH] = {……}; //儲存障礙物位置

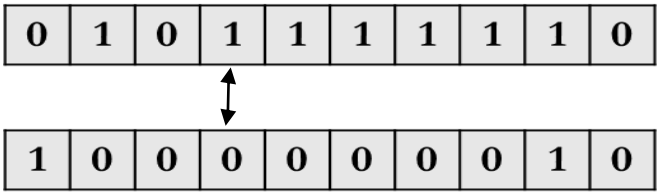
```
struct Gene
{
    bool genes[GENETIC_LENGTH];
    int fitness;
};
```

適應函數

```
void fitness_calculate(Gene input)
{
    int fitness = 0;
    if(路徑不會與障礙物重疊){ fitness+=200; }
    if(恰一個邊與起點相連){ fitness+=50; }
    if(恰一個邊與終點相連){ fitness+=50; }
    if(邊形成的路徑相連接){ fitness+=50; }
    fitness+=路徑加總 distance;

    Gene.fitness = fitness;
}
```

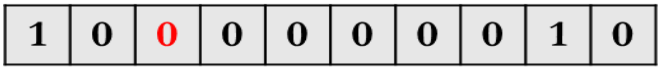
交配

將 genes[i]做隨機單點交換



突變

將 genes[i]做隨機單點突變

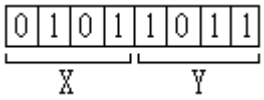


(五)五子棋

基因編碼:

每個基因表示下一手棋子所下的位置

長寬 16X16 的棋盤

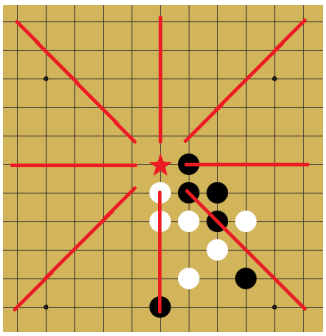


基因前半的 bits 轉換成數字即 X 軸的座標後半表示 Y 軸座標

適應函數:

	加權分數
五連	10^8
阻擋對手五連	10^7
阻擋對手四連	10^6
四連	10^5
阻擋對手三連	10^4
一個以上三連	10^3
一個三連	10^2
二連	10^1
該座標沒有棋子	10^0

檢查基因表示的座標周圍 8 個方向,5 個棋子的距離依照加權分數增加分數



int N=棋盤長寬

struct Gene

{

bool gene[$\log_2 N * 2$];

int 分數;

}

```
BEGIN
Initial population;
Fitness;
//產生 200 代基因並挑選出最高分的基因當作下一手棋的座標
for(i=1;i<=200&最高分的基因分數<五連的加權分數;i++) {
   Selection;
   Crossover;
   Mutation;
   Fitness;
}
END
```

Initial population:

亂數隨機產生 40 五子棋基因

Fitness:

```
if(該座標沒有棋子) {
   分數+1;
   //檢查 8 個方向並依加權分數加分
   分數=分數 + weight_function(檢查上方連續 5 個棋盤座標的情況);
   分數=分數 + weight_function(檢查右上方連續 5 個棋盤座標的情況);
   分數=分數 + weight_function(檢查右方連續 5 個棋盤座標的情況);
   分數=分數 + weight_function(檢查右下方連續 5 個棋盤座標的情況);
   分數=分數 + weight_function(檢查下方連續 5 個棋盤座標的情況);
   分數=分數 + weight_function(檢查左下方連續 5 個棋盤座標的情況);
   分數=分數 + weight_function(檢查左方連續 5 個棋盤座標的情況);
   分數=分數 + weight_function(檢查左上方連續 5 個棋盤座標的情況);
}
int weight_function(連續 5 個棋盤座標的情況) {
   if(五連) {return 10^8;}
else if(阻擋對手五連) {return 10^7;}
else if(阻擋對手四連) {return 10^6;}
else if(四連) {return 10^5;}
else if(阻擋對手三連) {return 10^4;}
else if(一個以上三連) {return 10^3;}
else if(一個三連) {return 10^2;}
else if(二連) {return 10^1;}
}
```

Selection:

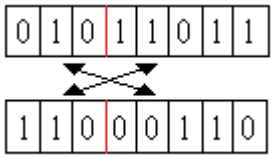
選擇分數較高的 20 組基因複製

分數低的淘汰

Crossover:

隨機找一個 bit 為分段的點進行交配

例如第 3 個 bit:



Mutation:

從母群體中隨機找一個基因隨機挑一個 bit

如果是'0'改成'1'如果是'1'改成'0'

(六)Wash machine

有若干種衣服

每種衣服有不同的重量，且每種衣服有若干件

如：

棉上衣：100g，5 件

大衣：500g，2 件

牛仔褲:200g，3 件

如果每次洗衣機最多可以洗 700g 的衣服，最少可以幾次完成

基因列以每 16bit 表示一件衣服的重量

0000000111110100	0000000001100100	0000000011001000	……
500g 大衣	100g 棉上衣	200g 牛仔褲	

fitness function:

先取得總重

總共桶數(這串基因列的排序方式最後需要幾桶)

最低桶數(總重 / 每桶限制)

誤差桶數(總共桶數 - 最低桶數)

誤差重量(每桶限制 - 實際重量)

1. 100 - ((誤差桶數 / 最低桶數) * 100)
   (取以最低桶數為標準差，誤差一個標準差內的基因列
2. (100 / 總共桶數) * (1 - 誤差重量/每桶限制)
   (該基因串的每一桶都要做一次，如果全部都是最滿就有 100 分

如果算出來小於零都以零表示

計算桶數：

```
int cal_number(基因列)
{
    將基因列以每 16bit 為單位切割成陣列
    while(基因陣列元素存在)
    {
        if(該桶重量 + 元素重量 > 每桶限制)
        {
            合計桶數++
            該桶重量歸零
```

```
        }
        if(該桶重量 + 元素重量 <= 每桶限制)
        {
            該桶重量 += 元素重量
            跳到下一個元素(continue)
        }
    }
    return 合計桶數
}
```

計算桶重：

```
int cal_weight(基因列 , 第幾桶)
{
    切割並尋找到所求之桶數
    return 該桶合計重量
}
```

計算權重：

```
double fitness()
{
    總共桶數 = cal_number(基因列)
    最低桶數 = 總重 / 每桶限制
    誤差桶數 = 總共桶數 - 最低桶數
    for(每桶做一次)
    {
        誤差重量[] = 每桶限制 - cal_weight(基因列 , 第幾桶)
        權重值 += (100 / 總共桶數) * (1 - 誤差重量/每桶限制)
    }
    權重值 += 100 - (( 誤差桶數 / 最低桶數 ) * 100)

    return 權重
}
```

交配:

以一次 16bit 為一單位進行互換

0000000111110100	0000000001100100	0000000011001000	……

0000000001100100	0000000111110100	0000000111110100	……