使用 Tabu 搜尋法自動找尋程式錯誤

許銀雄　何祖鳳　尹邦嚴　卓輝龍
銘傳大學資訊管理研究所
333 桃園縣龜山鄉大同村德明路五號
hsong@mcu.edu.tw　tfho@mcu.edu.tw　pyyin@mcu.edu.tw　richardcho@ms13.url.com.tw

摘要

產生測試資料以找出程式錯誤是軟體計劃中重要而且費時費力的一個項目。如果測試資料可以有效的自動產生，除了可以節省人力經費之外，更可以提昇軟體的正確性。在本論文中，我們介紹一種將軟體測試資料產生轉換為資料搜尋的架構，並利用 Tabu 搜尋法來進行資料搜尋，亦即，自動找出會讓程式出現錯誤的測試資料。我們會展示幾個小程式的測試結果，並探討可以採取什麼策略來利用 Tabu 搜尋更快速且有效的找到軟體的錯誤。

關鍵詞：軟體測試，軟體自動測試，Tabu 搜尋法。

一、緒論

軟體測試是軟體系統開發過程中重要且耗費成本的一個程序。即使並沒有增加任何的系統功能，卻經常是要耗費 50%以上的整體成本[2]。但是為了確保軟體的品質，它卻也是無可或缺的一個重要程序。如果可以讓這個程序以比較自動化的方式來進行，一方面可以大幅節省成本，另一方面也可以提高軟體的品質，所以就有了自動化測試的概念，有不少相關的研究相繼被提出[7, 9, 10, 16, 17]。而 Ould 也指出這方面的進展將會是軟體測試重要的一個方向[12]。

軟體測試過程中，最主要的工作是要產生適當的測試資料。依照測試軟體是否被執行，我們可以將測試資料產生的方法分成靜態以及動態兩種[17]。靜態的方式所指的是測試資料的產生並不需要去執行受測軟體。其中最常用的方式就是利用代入符號推導(symbolic execution)[3, 6]，以找出所須資料的限制條件，之後再嘗試找出符合這些限制條件的解答的方式。因為程式本身的特性以及複雜程度，這類的方式在很多情形下並無法執行，例如：迴圈(loop)、遞迴(recursive)、動態資料結構等等。而動態的方式則是指在產生測試資料時，需要執行受測軟體的情形。這類的方式，通常的做法是把測試資料產生看成是一個資料搜尋的問題：利用特殊的資料搜尋方法，找出測試資料，將此資料餵入受測軟體執行，再根據執行結果找出下一個可能更好的測試資料的一個循環過程[4,11]。本論文所採取的方法即是動態的方法。

另外，從測試資料產生時是否需要看到程式的架構以及內容來分類，測試方法可以分成白箱以及黑箱測試兩類[13, 15]。白箱測試最主要是允許測試人員審查程式內部結構，依其邏輯架構來設計測試案例。這樣的做法因為可以瞭解到程式的內容，有比較多的資訊，理論上可以設計出較適當的測試案例。其代價就是要對程式的內容進行分析。而黑箱測試則是不去審視受測程式的內部結構，直接根據程式規格(program specification)所描述的功能來進行測試。相較於白箱測試，黑箱由於資訊較少，所以可能會比較難很快找到適當的測試案例，但它卻比較單純而且直接。為了要讓黑箱測試有效率的實施，必須要有好的搜尋以及指

引技巧來支援。本論文是採用黑箱測試的方式，並採用 Tabu 搜尋法[5]的最佳化技術來導引它快速找到適當的測試資料。

Tracey 等人提出動態測試的架構，以 SPARK-Ada 來描述軟體的規格，並利用模擬退火法來進行軟體自動測試[16]。本論文採用與他們同樣的架構，但是改採 Tabu 搜尋法來實現，並就 Tabu 搜尋法在軟體自動測試上所可能採用的策略進行探討。

本文將在第二節中介紹軟體自動測試的概念，第三節中說明 Tabu 搜尋法，第四節對幾個實例進行探討，第五節說明 Tabu 搜尋法運用時的策略，最後則是結論。

二、軟體自動測試概念

軟體測試最主要的目的就是要找出軟體的錯誤，亦即找出可以讓程式得到錯誤結果的測試案例。在黑箱測試中，即是要找出不符合程式規格的測試案例。所以要證明一個程式有錯誤，我們只需要找到一個不符程式規格的測試案例便算是成功了。在本論文中，我們採用 Tracey 等人所使用的 SPARK-Ada[1, 16]來描述程式的正規程式規格(formal specification)。它的格式大致如圖一所示，它主要包括前置條件(pre)以及後置(post)條件兩個部分。

```
procedure Int_Root(N : in Integer , Root : out
Integer)
--## pre     N >= 0
--## post    (Root * Root <= N) and
--##         (N < (Root + 1 ) * ( Root + 1 ) ) ;
```

圖一、規格正規化[16]

前置條件所描述的是輸入變數的條件，而後置條件所描述的是這個程式執行之後必須要符合的條件。前置條件中的變數是輸入的變數，而後置條件中的變數可以是輸入也可以是輸出的變數。而對於會被程式改變的輸入變數，我們加入~符號表示執行後的變數結果(例如：N~代表程式執行後的 N)。以圖一為例，我們可以把它解讀成是當我們輸入一個大於 0 的 N，程式執行後得到的 Root 必須符合 Root*Root<=N 以及 N <(Root+1)*(Root+1)。

在規格建立的之後，接下來我們要做的就是找尋輸入資料，去測試是否符合程式的規格。所以針對內含錯誤的程式，如果我們找到一個可以讓前置條件符合，但執行後的結果卻會無法符合後置條件的測試案例，便表示這個程式有錯誤，也就達到我們的目的。這一部份是最花時間的部分。但是如果將產生輸入資料的過程自動化，並且利用程式來控制資料的變化情況，使得資料的產生可以越來越逼近某一個值，而這個值可以讓我們知道程式錯誤的發生，那麼就可以讓軟體測試自動執行。接下來我們先來說明如何讓這個程序可以進行。

動態方法

為了要讓動態方法可以執行，我們要定義一個目標函數(objective function)，以便利用它來導引我們的搜尋。這個目標函數是要用來幫助我們判別我們找到的測試資料的好壞，對於離我們所要的測試資料附近的測試資料，它應該要給予比較小的值，對於離比較遠的測試資料，它應該要給予比較大的值，對於可以讓我們找到程式錯誤的測試資料，它應該要給予 0。如此一來，我們就可以把問題轉換成為一個找最小目標值的最佳化問題。舉例來說，如果 X>100 是找到程式錯誤所要符合的限制式，對於輸入 X=98 與 X=16，我們可以假設 X=98 應該會比 X=16 更為接近我所想要的結果，因為|100-98| < |100-16|。所以 X=98 所得到的值應該要比 X=16 來得小。

目標函數的計算是依照下列的步驟計算而成，主要是要將正規程式規格轉換成 DNF (Disjunctive Normal Form)的形式。首先，是將前置條件以 AND (∧)和後置條件的相反

(negative)結合。例如，A → (B ∧ C)轉換成 A ∧ ~ (B ∧ C)，亦即，若符合前置條件 A 但不符合後置條件(B ∧ C)，則表示程式有錯誤，所以我們的目的就是要找出讓 A ∧ ~ (B ∧ C)符合的輸入資料。之後，這個式子在被轉換成為 DNF 的形式，亦即，所有的配對(pair)之間以 OR (∨)結合，而每個配對的項目(term)則是以 AND (∧)結合。例如，上一個式子就轉換成 (A ∧ ~B) ∨ (A ∧ ~C)的形式。只要符合任何一個配對，其實就表示程式有錯誤。對於一個配對，它的任何一個項目都依照表一的計算方式來計算成本，而所有項目的成本都加總成為這一個配對的總成本[16]。其中的 K 是一個常數，當條件不符合時都會被加到成本裡面。

表一、成本函式計算[16]

項目	成本值
Boolean	If True then 0 else K
a=b	If abs(a-b) = 0 then 0 Else abs(a-b)+ K
a≠b	If abs(a–b) ≠ 0 then 0 else K
a<b	If a–b < 0 then 0 else (a-b) + K
a≦b	If a–b≦0 then 0 else (a-b) + K
a>b	If b-a < 0 then 0 else abs(b–a) + K
a≧b	If b-a≦ 0 then 0 else abs(b-a) + K

透過這個目標函數，我們可以利用它來作為 Tabu 搜尋時的導引，導引我們往可能導致程式錯誤的測試資料前進。

簡單範例

我們以一個 Tracey 使用過的簡單例子來說明目標函數的用法[16]。圖二是一個程式的規格。這個程式會把 0 到 9 的數加 1，但是當輸入的數是 10 的時候，則又會回到 0。

我們首先要把規格轉換成 DNF，前置條件本身已是 DNF 的形式，而後置條件可以先轉換成 DNF 的形式之後在與前置條件結合。之後，可以得到以下兩個配對：

(N>=0) ∧ (N<=10) ∧ (N<10) ∧ (N˜ ≠ N+1) (1)

(N>=0) ∧ (N<=10) ∧ (N=10) ∧ (N˜ ≠0) (2)

```
procedure Wrap_Inc(N:in out Integer);
--#pre    N >=0   and   N<= 10;
--#post   (N<10→N˜=N+1) and (N=10→N˜=0 );
```

圖二、Wrap_Inc 規格

以第二個配對來看，如果此時 N 為 3，而假設執行後的 N˜ 是 4，則它各個項目的成本如下所示：

項目	成本值
N ≧ 0	0
N ≦ 10	0
N = 10	abs(3-10)+ K
N˜≠ 0	0

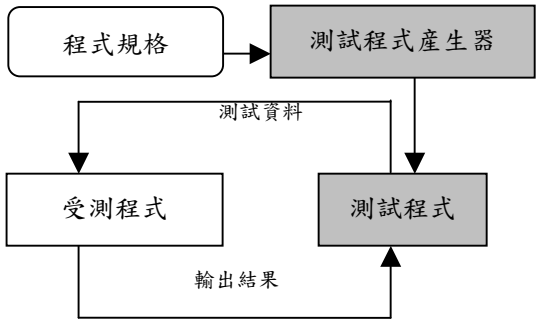
所以 N 為 3 對於這個配對的總成本不等於 0，亦即這個配對並不成立，所以也就是 N 為 3 沒有為這個配對找到程式的錯誤。如果 N=10，而且假設程式有錯所以執行後的 N˜是 11(例如，程式本來應該寫 N<10，但卻寫成 N<=10 的情形)，各個項目所貢獻的成本如下所示：

項目	成本值
N ≧ 0	0
N ≦ 10	0
N = 10	0
N ≠ 0	0

所以 N=10 時，這個配對的總成本為 0，代表它會使這個配對成立，亦即是 N=10 時會導致這個受測程式發生錯誤。這代表著成功的找到一個可以讓受測程式發生錯誤的測試資料。這個錯誤的程式附在本論文的最後。

自動化程式測試系統的架構

有了目標函數的定義，接下來我們來說明一個完整的自動化程式測試系統的架構。圖三所示即為系統的架構。一個受測程式會有一份對映的程式規格。測試程式產生器會剖析(parse)程式規格，之後產生一個與此規格相對映的測試程式。而測試程式裡面有一個最佳化(optimization)工具組，它會不斷的產出測試資料，餵給受測程式去執行，之後再根據執行結果判斷是否已經找到程式的錯誤。若不是，則會找下一組測試資料繼續進行測試，如此不斷重複的實施，直到錯誤找到，或停止條件達到為止。而在找下一組測試資料時，需要有一個好的最佳化工具組來幫忙，以便在找出的測試資料會比上一次好。Tabu 搜尋法是一個被廣泛使用的最佳化工具，所以本論是利用它來幫忙找下一組測試資料。下一節將對 Tabu 搜尋法進行說明。



圖三、自動化程式測試系統之架構

三、Tabu搜尋法簡介

Tabu搜尋法是Glover所提出的一種泛導引式搜尋法(meta-heuristic)[5]，它已經被成功的應用到許多不同的領域中，包括；排程問題、圖形問題、資源分配問題、等等[8, 14, 18]。它是屬於區域搜尋法的一支但是它的搜尋卻是以一種導引的方式進行。它的導引方式是透過歷史資料的維護來進行探索(exploration)，以克服只能找到區域最佳解的問題。

Tabu搜尋法的做法是，先從一個初始解開始，然後開始從一個解跳到另一個解，並在跳到另一個解的時候，利用記錄先前走過的路徑的方式禁止沒有意義的往回走的方式來避免迴圈的發生。它利用一個tabu名單(tabu list)來記錄先前走過的路徑，tabu名單可說是Tabu搜尋法的記憶。而在從一個解跳到另一個解時，它會找尋其最優的鄰近解作為下一步的移動路徑；若這個鄰近解優於目前之解，則跳到該鄰近解，以這種方式來逼進最佳解；而如果該鄰近解並沒有比目前的解好，則只有在它沒有被禁止的情形下才會往前跳(亦即，沒有在tabu名單之中)。圖四是Tabu搜尋法的演算法。

```
Tabu search
{
   set initial solution and global_cost
   while (!stop())
   {
     generate n neighborhoods based on current
        solution, s1, s2, …, sn   (probability)
     sort these n solutions according to cost
        function, sa[1] ≤ sa[2] ≤ sa[3] ≤…≤ sa[n]
     for ( i=1; i<=n; i++)
     if (   cost (sa[i]) < global_cost    )
     { /* perform improving strategy */
       accept sa[i]
       global_cost = cost(sa[i])
       current_solution = sa[i]
       current_cost = cost (sa[i])
       insert to tabu list
       break
     } else /* perform jump-out strategy */
       if (sa[i] not in tabu list)
       {
         accept sa[i]
         current _cost = cost (sa[i])
         current _solution = sa[i]
         insert to tabu list
         break
       }
   }
}
```

圖四、Tabu 搜尋演算法

由圖四得知，從原始的可行解開始， 在求解的每一過程中， 我們都先到鄰近解中去選取最好的鄰近解作為變動方向的路徑(也就是最有利於目標函數者)變更， 逐步運作逼近

最佳解，直到符合結束規則為止。

Tabu 的組成要素

Tabu搜尋法運作基本模組概念可分為五個：移步、tabu名單、淩駕規則、起始解與停止準則，各分述如下[5]：

1. 移步(move)：從一個可行解到下一個可行解的過程，就叫做移步，而移動的過程將滿足移步的屬性，也就是向最好的鄰近解移動。

2. tabu名單(tabu list)：為了知曉可行解的好壞，Tabu搜尋法使用短期的記憶架構(short term Memory)，主要目的是為了避免進入到搜尋的迴圈狀態，我們將設定在某一個次數內，禁止反向的移步。

3. 淩駕規則(aspiration criterion)鄰近解若在tabu名單內，但是優於現存之最佳解時，我們便接受此禁忌移步。

4. 起始解(initialization solution)：由最原始的可行解中，所產生一個起始解。

5. 停止準則(stopping criterion)：停止搜尋的準則。

四、實例探討

在本節中，我們將介紹幾個利用Tabu搜尋法來實作時所使用的例子以及執行的結果。在介紹這些例子之前，我們先說明一下，我們在實驗中Tabu搜尋法所使用的策略。

首先是Tabu搜尋法的移步；它是要由所有可能的鄰近解中，選取最優者作為改善的移動路徑。所以如何設計移步，以便找到比較好的鄰居是一個關鍵的步驟。除了找鄰近點之外，由於程式的輸入值的可能範圍可以非常之大，如果只找鄰居，可能要很久才能跳到最佳解，為了對付這樣的情況，本論文為每個輸入變數的移步設計四種移步，分別是：往前走一小步、往後走一小步、往前走一大步、以及往後走一大步等。有一個比率R代表採取小步或大步的比率。R的值愈高表示走小步的機率愈高。另外，tabu名單的設計，本論文中是採用一個queue來記錄它的暫存記憶。由於它是暫存記憶，所以queue的長度不會太長。當變數只有一個時，我們使用2，但當變數的個數比較多時，它的長度可以稍長。

測試的例子，我們直接利用 Tracey 等人所使用過的三個程式範例[16]，為了方便說明，我們還是把這三個程式的規格列出，並且把錯誤的程式列在論文的最後。這三個程式分別是 Middle，Bubble sort，以及 Tomorrow。

圖五是 Middle 這個程式的規格。這個程式會傳回三個輸入值中，大小介於中間的那個值回來。當其中有兩個輸入值是相同時，應該要傳回這個值。但是因為程式的撰寫錯誤，當兩個輸入值相同時，卻永遠只會傳回 A 的值。所以我們希望找的的測試資料應當是要三個輸入值中有兩個相同，但第三個不同的情形。

```
Function Middle (A,B, C : Integer) return
Integer
--#pre True ;
--#return M => ((B<A and A<C) → M=A) and
--#         ( (C<A and A<B) → M=A) and
--#         ( (A<B and B<C) → M=B) and
--#         ( (C<B and C<A) → M=B) and
--#         ( (A<C and C<B) → M=C) and
--#         ( (A<C and C<B) → M=C) and
--#         ( (B<C and C<A) → M=C) and
--#         (B=C→ M=B) and (A=C → M=A) and
            (A=B → M=A)
```

圖五、Middle 程式的規格

圖六是 Bubble sort 程式的規格。當我

們輸入一個包含 10 個範圍在 1…1000 的整數值的陣列時，Bubble sort 程式應該要把這個陣列的 10 個數值以遞增方式由小排到大。但是因為程式撰寫的錯誤，迴圈少做了一次，所以當最小值落在陣列的最後一個位置時，便會得到錯誤的結果。所以我們希望找到的測試資料是陣列中的最小值落在它的最後一個位置。

```
type My_Int is Integer range 1…1000
type My_Arr is array(1…10) of My_Int;
procedure Bubble_Sort (A: in out My_Arr);
--#pre    True;
--#post  X(1)<=X(2) and X(2) <=X(3) and
--#         X(3)<=X(4) and X(4) <=X(5) and
--#         X(5)<=X(6) and X(6) <=X(7) and
--#         X(7)<=X(8) and X(8) <=X(9) and
--#         X(9)<=X(10)
```

圖六、Bubble Sort 規格

圖七則是 Tomorrow 程式的規格。Tomorrow 程式主要是算出輸入日期隔天的日期。因為有閏年的因素，所以每到閏年時如果輸入是 2 月 28 日的話，則隔天的日期就要是 2 月 29 日，而不是 3 月 1 日。但是因為這個程式的撰寫錯誤，導致在月日為 2 月 28 日，而年為 100 的倍數但卻非 400 的倍數時候會出錯。而這也正是我們希望找到的測試資料。

本論文利用前述的做法分別對上述三個程式範例做了實驗。這個實驗是在 Pentiuem II 233MMX，64MB 記憶體，而作業系統為 Window2000 的機器上實施。要驗證一個程式是否正確，DNF 中的每個配對都必須驗證。對於不會檢驗出範例程式錯誤的配對，最後都會因為執行時間或次數超過而結束。為簡化問題，我們只列出因範例程式錯誤導致無法滿足的 DNF 的結果。

```
type Day_Type is
   (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
type Date_Type is range 1…31;
type Month_Type is
   (Jan,Feb,Mar,Apr,May,Jun,
    Jul,Aug,Sep,Oct,Nov,Dec);
Type Year_Type is range 1900…3000;
procedure Tomorrow
   (Day:in Day_Type;
    Date:in Date_Type;
    Month: in Month_Type;
    Year: in Year_Type;
    Next_Day: out Day_Type;
    Next Date: out Date_Type;
    Next_Month: out Month_Type;
    Next Year: out Year_Type;);
--#pre (Date = 31 →
--#   (Month = Jan or Month = Mar or
--#     Month = May or Month = Jul or
--#     Month = Aug or Month = Oct or
--#     Month = Dec) and
--#   (Month = Feb → Date <=29);
--#post (Day = Sun → Next_Day = Mon) and
--#   (Day /=Sun → Next_Day = Succ(day)) and
--#   (Date = 31 →
--#     (Next_Date = 1 and
--#       (Month = Dec →
--#         (Next_Month = Jan and
--#           Next_Year = Year +1)) and
--#       (Month /= Dec →
--#         Next_Month=Succ(Month)))) and
--#   ((((Date=30 and
--#     (Month=Apr or Month = Jun or
--#      Month =Sep or Month =Nov)) or
--#     (Date=29 and Month =Feb)) →
--#       (Next_Date = 1 and
--#         Next_Month = Succ(Month) and
--#         Next_Year =Year)) and
--#   (((Date = 28 and Month = Feb and
--#       ((Year mod 4=0 and Year mod 100 /= 0)
--#           or Year mod 400 = 0)) →
--#         (Next_Date=29 and
--#           Next_Month = Month and
--#           Next_Year = Year))and
--#   ((Date=28 and Month = Feb and not
--#     ((Year mod 4 =0 and Year mod 100 /=0)
--#         or Year mod 400 =0)) →
--#           (Next Date = 1 and
--#                 Next_Month = Mar and
--#                   Next_Year=Year))));
```

圖七、Tomorrow 程式規格

表二所列是這三個問題程式的測試資料的特性。其中參數空間指的是受測程式所有可能測試資料的個數。錯誤解個數指的是會讓該受測程式發生錯誤的測試資料個數。而錯誤解分佈是參數空間/錯誤解個數，它代表的是平均多少的解之中才會一個錯誤解。

表二、受測程式的測試資料特性

受測程式	參數空間	錯誤解個數	錯誤解分佈
Middle	1E+9	299800	3.3E+03
Bubble Sort	1E+30	1E+29	10.0
Tomorrow	2.86E+6	9	3.2E+05

表三是實驗的結果，它們都是執行了50,000 次後平均的結果。其中搜尋個數指的是Tabu 搜尋法平均找了多少個解才找到一個錯誤解。搜尋比率是搜尋個數/表二的參數空間，它代表整個參數空間被搜尋的比率。

比較表三的搜尋個數以及表二的錯誤解分佈，可以發現 Tabu 搜尋法有很好的效率。而其中 Bubble sort 的部分，比平均稍差，這是因為 Tabu 是採用機率的方式，且是採用區域搜尋的方式，所以對於很容易找到解答的問題，會需要暖身的動作。但是就執行時間來看，其速度真的是很快。

表三、實驗結果

受測程式	搜尋個數	搜尋比率	執行時間
Middle	153	1.5E-07	<0.01sec
Bubble Sort	624	6.2E-28	<0.01sec
Tomorrow	11712	4.1E-03	0.04sec

五、Tabu 搜尋法策略探討

本論文的實驗中，只用了前述簡單的策略來導引搜尋的進行。從結果來看，已經可以有不錯的效率。但是由於軟體有很大的彈性，它的參數空間(parameter space)會隨著參數個數而快速擴充。如此一來，測試資料的自動產生可能就會需要耗費龐大時間。面對這樣的問題，我們可以利用 Tabu 搜尋法中策略的調整來加速最佳化的進行。以下我們就幾個可能的策略加以探討。

首先，上一節中說明本論文採用的移步策略是每個變數會有四種移步：向前/向後跳一小步，以及向前/向後跳一大步。其中有跳一大步的移步其實就是為了要讓它可以有往遠處跳的可能性。至於決定近跳或遠跳，先前的實驗中是採用固定的比率，亦即在任何時候都相同。但是其實也可以採用動態的比率，亦即這個比率會隨著時間而變化。一個不錯的方式是隨著時間的前進，讓遠跳的比率逐漸降低，它的意涵是在剛開始的時候允許它做比較廣泛的探索，但是越到最後，則要它盡量做區域最佳解的探索。

上述的移步中，所謂的跳一大步或一小步，在上一節的實驗中，大步以及小步各有一個固定的距離，而實際跳的距離則是透過亂數乘以這個固定距離而求得。另一個不錯的做法是讓這個固定的距離也是動態的，讓它與目前解(current solution)的總成本成正比的關係。如此做的意涵在於如果總成本越高表示離目標越遠，所以應該跳遠一點，否則應該跳近一點。

六、結論

軟體測試是軟體發展重要且耗費成本的一個程序。將這個程序予以自動化不但可以節省成本而且可以提高軟體的品質。本論文將 Tabu 搜尋法應用到程式錯誤的自動搜尋。不論從搜尋結果或是執行速度來看都有很不錯的效果。相關的研究如果可以更加完整，並落實地將整個系統完整建置，不只對於學術研究有很大貢獻，對於需要大量人力資源且品質要求甚高的軟體業界，也將會有很大的貢獻。

從本論文實驗的結果來看，雖然已有不錯的效果，然而由於軟體的彈性以及變化都很大，如何再就實際的問題進行更深入的分析探討，仍然有很大的發展空間。如何尋找適用於軟體測試的有效搜尋策略以達到適用範圍更廣以及更好的效率也是很需要深入研究的問題。

七、錯誤程式碼範例

以下所列是四個 Tracey 等人原來所使用的範例程式[16]，我們把它改成 C 語言的形式，為了方便閱讀，所以將它列出來。

1. Middle

```
int Middle(int A, int B, int C)
{
    if ((A<B && B<C) || (C<B and B<A))
        return B;
    else if ((A<C && C<B) || (B<C && C<A))
        return C;
    else if ((B<A && A<C) || (C<A && A<B))
        return A;
    else
        return A;
        /* 錯誤，當有兩個數值相同時，應當
        要傳回該值 */
}
```

2. Bubble Sort

```
void bubble_sort (int X[])
{
    int i, j, temp;
    for ( i = 1 ; i < 8 ; i++ )
     /* 錯誤，原來應為 9 ，而不是 8 */
        for ( j = 1 ; j < 9 ; j++ )
        {
            if( X[j] > X[j+1] )
            {
                temp = X[j] ;
                X[j] = X[j+1] ;
                X[j+1] = temp ;
            }
        }
}
```

3. Tomorrow

```
void Tomorrow (Day_Type Day, Date_Type
Date,
    Month_Type Month, Year_Type Year,
    Day_Type Next_Day, Date_Type Next_Date,
    Month_Type Next_Monty,
    Year_Type Next_Year)
{
    Next_Day = Day; Next_Date = Date;
    Next_Month = Month; Next_Year = Year;
    if (Day == Sun)
        Next_Day = Mon;
    else
        Next_Day = Day_Successive (Day);

    if (Month == Dec and Date == 31)
    {
        Next_Date = 1; Next_Month = Jan;
        Next_Year = Year + 1;
    } else if (Date == 28 && Month == Feb)
    {
        if (Year % 4 == 0 && Year % 100 != 0)
            /* 錯誤，應該還要加上
                        || Year % 400 ==0 */
            Next_Date = 29;
        else
        {
            Next_Date = 1;
            Next_Month = Mar;
        }
    } else if (Date == 31 ||
        (Date==29 && Month==Feb)) ||
        (Date==30 &&
            (Month ==Apr || Month == Jun ||
             Month==Sep || Month==Nov)))
    {
        Next_Date = 1;
        Next_Month =Month_Successive(Month);
    } else {
        Next_Date = Date + 1;
    }
}
```

4. Wrap

```
int Wrap_Inc(int &N)
{
      if (*N>10) /* 錯誤，應該是 >= */
            *N = 0;
      else
            *N = *N+1;
}
```

參考文獻

[1] J. Barnes, High Integrity Ada: The SPARK Approach, Addison-Wesley, 1997.

[2] B. Beizer, Software Testing Techniques , Thomson Computer Press, 2nd edition,1990.

[3] L. Clarke, "A system to generate test data and symbolically execute programs", IEEE Trans. on Software Engineering, pp.215-222, 1976.

[4] M. J. Gallagher and V. L. Narashimhan, "ADTEST: a test data generation suite for ada software systems", IEEE Trans. on Software Engineering, Vol.23, No.8, pp.473-484, 1997.

[5] F. Glover, "Tabu search – Part I", ORSA J. Comput., Vol1, pp.190-206, 1989.

[6] J. King, "Symbolic execution and program testing", Comm. Of ACM, Vol.19, No.7, pp.385-394, 1976.

[7] B. Jones, H. Sthamer, and D. Eyres, "Automatic structuring testing using genetic algorithms", Software Engineering Journal, Vol.11, No.5, pp.299-306, 1996.

[8] K. Jornsten and A. Lokketangen, "Tabu search for weighted k-cardinality trees", Asia-Pacific Journal of Operating Research, Vol.14, No.2, pp.9-26, 1997.

[9] B, Korel, "Automated software test data generation", IEEE Trans. on Software Engineering, Vol.16, No.8, pp.870-879, 1990.

[10] B, Korel, "Auotmated test data generation for programs with procedures", Proceedings of ACM SIGSOFT international symposium on Software testing and analysis, pp.209-215, 1996.

[11] W. Miller and D. Spooner, "Automatic generation of floating-point test data", IEEE Trans. on Software Engineering, Vol.2, No.4, pp.223-226, 1976.

[12] M. Ould, "Testing – a challenge to method and tool developers", Software Engineering Journal, Vol.6, No.2, pp.59-64, 1991.

[13] R. Patton, Software Testing, Sams, 2000.

[14] J. Skorin-Kapov, "Tabu search applied to the quadratic assignment problem", ORSA Journal on Computing, Vol.2, No.1, pp.33-45, 1990.

[15] I. Sommerville, Software Engineering, 6th edition, Addison-Wesley, 2000.

[16] N. Tracey, J. Clark, and K. Mander, "Automated program flaw finding using simulated annealing", Proceedings of ACM SIGSOFT international symposium on Software testing and analysis, pp.73-81, 1998.

[17] N. Tracey, J. Clark, K. Mander, and J. McDermid, "Automated test-data generation for exception conditions", Software Practice and Experience, Vol. 30, pp. 61-79, 2000.

[18] M. Widmer, "The job-shop scheduling with tooling constraints: a tabu search approach", Journal of Operating Research, Vol.42, pp.75-82, 1991.