11

人工智慧

人工智慧（Artificial Intelligence; AI）就是機器執行我們認為需要智慧之工作的能力。比方說，在真實生活裡，當一名警察徒步追捕一個人時，他一定會想盡辦法抄近路。所謂的近路需視這名警察對他位在哪裡，要去哪裡，以及對追捕地區當地環境的了解而決定。在線上遊戲裡，主角要做到類似這種行為，可以用編寫遊戲程式的方式達成。這就是一種人工智慧的應用。

人工智慧的概念已經出現很長一段時間，早自十八世紀起，一些著名的哲學家就曾經爭論過機器會不會思考的問題。但人工智慧的概念受到一般社會大眾注意，是始自 1980 年代大型遊樂場遊戲的流行。直到 1997 年，IBM 超級電腦深藍（Deep Blue）挑戰西洋棋大師卡斯帕洛夫（Garry Kasparov）成功，人工智慧才真正引起一般人的關注。那次著名的六局四勝比賽中，卡斯帕洛夫落敗，人工智慧則寫下首度超越人類智慧的歷史。此後人工智慧即深入比西洋棋更複雜的遊戲中。目前一些即時戰略遊戲，像任天堂 Gamecube 的《皮克敏星球探險》系列和電腦遊戲《終極動員令：將軍》等，它們所用的人工智慧都令人嘆為觀止。

11 人工智慧

現在，再回頭來談我們要用的 Flash。無可否認，Flash 不可能用 ActionScript 寫出像深藍一樣程度的人工智慧。但是，它可以寫出相當好的程式碼，用來製作讓遊戲既有趣又好玩所需的人工智慧。當然許多遊戲並不需要用到人工智慧，像是多人玩的線上西洋跳棋（因為每名玩家都有大腦！）但是也有許多遊戲，既使簡單到像乒乓球或一個基本的平台遊戲，仍需要一定程度的人工智慧，讓玩家樂此不疲。這一章就要介紹 Flash 裡的人工智慧，以及遊戲裡使用人工智慧的趣味所在，和一些實際應用的範例。

11.1 人工智慧的類型

在你大概知道什麼是人工智慧以後，我們現在就來看看人工智慧可以用來做些什麼，人工智慧在遊戲裡倒底扮演什麼樣的角色。以下就是目前人工智慧在遊戲裡一些主要的用途。

尋找路徑：對所有的遊戲開發工程師，尤其是對人工智慧還陌生的程式設計師而言，這是一項最重大的課題。尋找路徑，顧名思義，就是尋找一個點到另一個點之間的路徑。在「暗黑破壞神」（或上一章範例 iso_world.fla）之類的遊戲裡，玩家只要按到某個點，角色就會走到那個點。當走到那個點的路上將碰到一個物件時，角色會繞過物件繼續前進。在大部份這類先進的遊戲裡，角色所走的整個路徑，其實是從玩家按到那個點的那一刻開始計算（而不是從角色走出第一步時才計算）。

同樣地，尋找路徑的技術用在遊戲中的敵人時，一個怪獸可以被設計成跳過那個物件向角色直奔。

目前業界有許多類型的尋找路徑運算法，但所有權威遊戲開發者一致公認最好的一種是「A*」（A Star）。本章最後一節將討論 A* 運算法。

產生關卡：某些遊戲會在運行過程中建立隨機但智慧型的關卡吸引玩家。比方說，當玩某個遊戲兩次時，關卡的建構（如房間和牆），可能第一次和第二次都一樣，但敵人和神秘項目的位置第二次可能會不同。或者，整個關卡

都可能完全改觀，好像本章將介紹的迷宮遊戲，這類關卡就是用人工智慧自動產生。但用 AI 產生關卡的做法並不是慣例，有時候可能流行一陣，有時候卻只在某些特定的遊戲裡才看得到。

設計敵人行爲：遊戲裡的敵人使用尋找路徑技術，可以找到角色所在的位置。但當敵人到了角色面前會做出什麼舉動？是拿一把劍刺它？還是臨陣脫逃？還是跟角色聊聊？這是根據控制敵人舉動的人工智慧選項所決定。對全球發行的超大型遊戲來說（至少我聽說），這種人工智慧可能算是最複雜的元件之一。但是 Flash 裡這類角色扮演遊戲的人工智慧卻簡單得多，而（與大型遊戲相比）Flash 遊戲也相對簡單得多。

建立神經網路：神經網路是一個會學習的人工智慧，它會根據遊戲進行中現場即時估算的內部數值參數產出機械反應，而在不同場合做出不同行為。這個概念是近年才開始被用到遊戲設計上。像一些玩家與 AI 對抗的戰略遊戲裡，雙方都配備有飛機、坦克、軍艦和士兵。假設玩家不斷用飛機攻擊敵軍，人工智慧就會從這個經驗中學習。它就會想：「嘿！這樣下去不是辦法！我要採取行動阻止飛機繼續攻擊。哈！有了！我就派出坦克去轟掉那些飛機！」這種人工智慧就非常人性化。它會辨識行為模式，判斷行為模式對自己不利，然後採取行動，破壞這種行為。神經網路的應用在電子教學上也被大量採用。例如一個教人觸摸打字法的軟體，當使用者故意打一個錯字開玩笑時，它的人工智慧可能已被設計成會處罰使用者，或當軟體偵測到使用者同時敲打所有的按鍵時，它的人工智慧可能會叫使用者該休息一下。

AI 要有輸有贏

有件事非常重要需要特別注意，就是一個人工智慧要能在遊戲裡發揮最佳功效，應該有不同等級的難易度。比方說，假設一個西洋棋遊戲裡，玩家對抗的是深藍，那玩家顯然是輸定了。可是，遊戲能讓玩家覺得好玩，最好的就是玩家有機會贏，而且最好是讓玩家知道有機會贏，那一旦玩家贏了就會有一種成就感。因此遊戲的難易度必須取得一個平衡。不過，無論如何，人工智慧的設計還是要盡可能做到最佳。因為設計到最佳程度的 AI，隨時還是有辦法讓它變笨，或讓遊戲有時用有時不用 AI，但要在 AI 製作完成後讓它變得

更聰明就相當困難了。比方說，為一個西洋跳棋遊戲建立了一個絕對不會輸的 AI 之後，要把遊戲變得簡單些，只需要讓電腦在選擇下一步時，採用隨機取樣方式，不使用人工智慧。這樣做雖然讓人工智慧不是那麼厲害，可是卻能讓遊戲比較好玩。但倒底要讓人工智慧笨到什麼程度？那只能由玩家玩一玩試試才能知道。

製作輪流玩的遊戲：人工智慧可以用來製作輪流玩的遊戲像西洋棋或西洋跳棋，市面上有許多不同程度的 AI 可用來製作這類遊戲。主要的兩種這類 AI，一種是根據當前棋盤上的狀況，決定下一步如何走；另一種是根據棋譜，策劃戰略，決定下一步。網路上就可以找到許多像西洋棋、西洋跳棋、「四小龍（Connect Four；火柴棒遊戲）」和英文拆字遊戲 Scrabble 等的人工智慧運算法（不過用 ActionScript 寫的並不多）。

自訂邏輯：前面所提到的每一種 AI 都可以自訂邏輯。其實在遊戲裡使用 AI 可以控制任何東西，從控制角色行為、控制顏色、音量、回音、速度到控制難易度。在光碟裡 Chapter11 下有一個基本範例，那是一個有人工智慧的乒乓球遊戲，其中對方的球拍聰明到會跟蹤球的移動擋球。而你也可以建立一個人工智慧，讓遊戲在某種情況下發生某件事。比方說，一個賽車遊戲裡，當玩家玩得太好（而你也想給玩家一點顏色看看）時，可以製造一場暴風雨讓遊戲變得比較難玩。

本章以下部份將探討自訂人工智慧、使用 AI 產生隨機但完美的迷宮，以及 A* 的尋找路徑。

11.2 手工打造人工智慧

這一節我們要先撇開人工智慧運算法，只討論如何自製人工智慧。但本節所提供的資訊，只涉及特定的人工智慧，連學習如何建立 AI 的皮毛都談不上。學習如何建立人工智慧需要看一些更專業、篇幅更長的書。

請打開光碟裡 Chapter11 的 shark_attack.swf 檔。這個檔看起來很像第七章「區塊世界」的 *Shark Attack!*（見圖 11.1），它是一個內有一些人工智慧敵人的區塊世界遊戲。這個遊戲是我為一家叫 Simply Scuba 的公司所製作。玩家的主角是一條紅魚，遊戲的目標是拿到鑰匙後走到門那裡，主角在行進過程中吃到硬幣或物件都會加分，而主角的敵人就是鯊魚。請連按兩下這個 SWF 檔打開它，玩上幾關（裡面只有四關），注意看看鯊魚的行為。它們都由一個相當簡單但有效的人工智慧所控制。這一節要學的就是如何製作一個像這樣簡單的人工智慧。

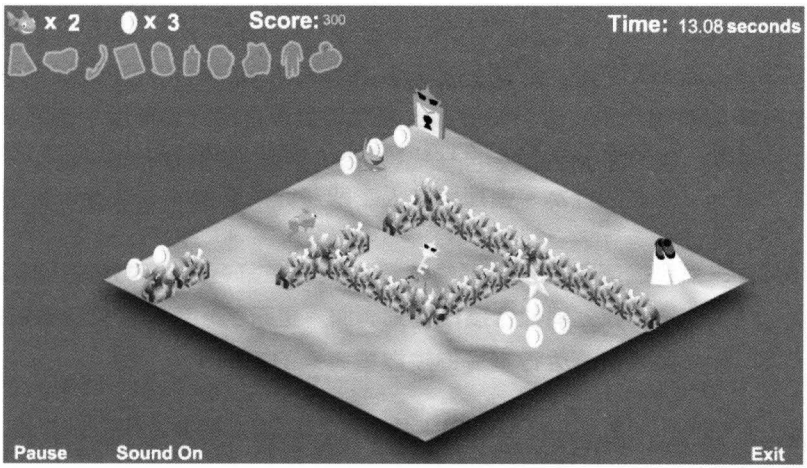


圖 11.1
Shark Attack!

11.2.1 控制角色的規則

學到這裡，你其實已經準備好完成一個自己的遊戲了。

請打開光碟內 Chapter09 的 run_away.fla 檔。這個檔裡有另外一個半成品遊戲「盜墓者（Grave Robber）」（見圖 11.2）內敵人的 AI，我們要看的就是這些敵人（這裡也叫「壞蛋」）如何動作。這些壞蛋都是僵屍，而（像小偷一樣好的）好人是一個活人，它由玩家控制。遊戲情節是主角到處找墓去盜，但僵屍每次都會出現要抓盜墓者。遊戲裡有牆，主角不能穿過牆。這個檔裡，主角和壞蛋間不需要偵測碰撞，因為我們只是舉例說明行為的 AI。請注意：這個檔（以及這一章所用的每一個範例檔）使用的都是區塊，所以如果你對區塊世界不熟，最好花

點時間去看一下（第七章）。同時，我們將探討這個檔內 AI 所用的 ActionScript，但不討論如何建立區塊世界，或主角與牆的碰撞等。

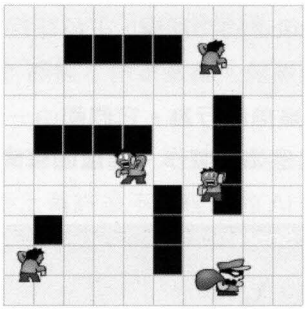


圖 11.2
盜墓者。

請用測試影片（Test Movie）看看這個遊戲。先到處移動一下主角，試探敵人僵屍的行為。這裡用的人工智慧非常類似 *Shark Attack!* 裡用的。

這個檔的程式碼中，有一段會在第一時間改變敵人僵屍運動的方向。為了精簡，我就稱執行這段程式碼叫「更新」。我們就來看看這段更新程式碼內，僵屍遵循以產生行為—追蹤盜墓者—的規則。

- 在任何時間，角色的移動限定為非水平，即垂直。
- 更新程式碼會檢查主角與敵人的相對位置，並根據下列規則儲存資訊：

  水平移動

  若主角向敵人左邊移動，則為 -1

  若主角與敵人在同一行移動，則為 0

  若主角向敵人右邊移動，則為 1

  垂直移動

  若主角在敵人上方移動，則為 -1

  若主角與敵人在同一列移動，則為 0

  若主角在敵人下方移動，則為 1

- 若水平值與垂直值均為 0，敵人就在主角同一區塊內，則更新程式碼不會改變敵人移動路徑。
- 若水平值或垂直值任一為 0，則更新程式碼即改變敵人移動方向。比方說，若水平值為 -1，垂直值為 0，則程式碼即知道主角是在敵人同一行的左邊，它就叫敵人向左邊移動。
- 若水平值和垂直值均非 0，則更新程式碼隨機選擇水平或垂直，讓壞蛋跟著移動。例如，若垂直值為 -1，水平值為 1，則人工智慧即知道主角位在敵人的右上方某處，它就隨機選擇水平或垂直移向主角。
- 更新程式碼內有一個隨機取樣條件（就是一個 `if` 條件陳述式）。當隨機取樣時，更新程式碼會不顧遊戲盤上當時的情況，完全隨機選擇一個方向移動。這個就是讓 AI 變笨的功能。而頻繁的隨機取樣，會讓人工智慧的行事不可預料。

在了解執行更新程式碼的邏輯之後，再來看看執行更新程式碼的時機。以下是更新程式碼執行的條件：

- 當敵人碰到一堵牆或任何阻擋物件時。
- 當影格最新的 `maxtime` 值傳遞出去後，每名敵人的 `maxtime` 值都會不一樣。

請匯出 SWF 檔測試一下這個人工智慧。請執行人工智慧，並保持開啟。在一段短短的時間內，僵屍最後會走到主角大致相同的區域內。

11.2.2 缺陷及解決之道

在看這個人工智慧用的 ActionScript 之前，我要提一下這個 AI 的缺陷。你可能注意到，這個人工智慧最好用在有許多牆的世界—幾乎是迷宮遊戲那樣的環境。因為敵人通常會待在非常靠牆的位置。由於敵人有這種行為模式，因此假設遊戲世界相當空曠，敵人就會常待在靠近遊戲世界邊界的地方。如果你還算喜歡這個人工智慧，但又希望讓它比靠牆行為傾向更聰明些，以下有幾個辦法可以做到：

- 改寫更新程式碼，讓角色先向區塊中心點靠攏再繼續運動，而不是先靠牆。這不只是為了讓畫面更好看，而且也是為了讓畫面更聰明。
- 當角色碰到牆時，讓轉向的優先順序高於去撞牆。目前這個 AI 會讓角色在走開前先撞幾次牆。由於碰撞發生非常快速，所以看起來並不十分明顯，但它確實出現這種情況。
- 增加斜向（對角線）運動。

11.2.3 敵人的 ActionScript

現在就來看看這個人工智慧的 ActionScript。它會在每個影格裡呼叫 baddyAI() 函式，這個函式會迴圈一個敵人陣列，研判當時是否該更新。如果是，就執行更新程式碼。

```
1     function baddyAI() {
2          for (var I = 0; i<game.baddies.length; ++i) {
3               var ob:Object = game.baddies[i];
4               ++ob.time;
5               var cell_x:Number = Math.ceil(ob.x/game.cellWidth);
6               var cell_y:Number = Math.ceil(ob.y/game.cellWidth);
7               var cell_over:Object = ?game.tiles[cell_x][cell_y];
8               var cell_x_temp:Number =
                ➔Math.ceil(ob.tempx/game.cellWidth);
9               var cell_y_temp:Number =
                ➔Math.ceil(ob.tempy/game.cellWidth);
10              var cell_over_temp:Object =
                ➔game.tiles[cell_x_temp][cell_y_temp];
11              if (!cell_over_temp.empty || ob.time ==
                ➔ob.maxtime) {
12                   ob.time = 0;
13                   ob.maxtime = 30+random(30);
14                   ob.tempx = ob.x;
15                   ob.tempy = ob.y;
16                   var tempDir = ob.dir;
17                   var xmov:Number = 0;
18                   var ymov:Number = 0;
19                   var speed:Number = Math.abs(ob.speed);
```

```
20                  var xsign:Number = (game.char.x-
                    ➔ob.x)/Math.abs((game.char.x-ob.x));
21                  var ysign:Number = (game.char.y-
                    ➔ob.y)/Math.abs((game.char.y-ob.y));
22                  if (random(10) == 0) {
23                       xsign = -1*xsign;
24                       ysign = -1*ysign;
25                  }
26                  if (xsign == ysign || xsign == -ysign) {
27                       var ran:Number = random(2);
28                       if (ran == 0) {
29                            xsign = 0;
30                       } else {
31                            ysign = 0;
32                       }
33                  }
34                  if (xsign != 0) {
35                       ymov = 0;
36                       xmov = xsign*speed;
37                       if (xmov>0) {
38                            var dir:String = "right";
39                       } else {
40                            var dir:String = "left";
41                       }
42                  } else if (ysign != 0) {
43                       xmov = 0;
44                       ymov = ysign*speed;
45                       if (ymov>0) {
46                            var dir:String = "down";
47                       } else {
48                            var dir:String = "up";
49                       }
50                  }
51                  ob.dir = dir;
52                  ob.clip.gotoAndStop(dir);
53                  ob.xmov = xmov;
54                  ob.ymov = ymov;
55             }
56        }
57   }
```

這個函式非常長，不過別擔心—裡面有很多資訊都是一再重複。最主要因為裡面有好幾個 `if` 陳述式，而且為 *x* 方向做的，都還要再為 *y* 方向重複一次。就像本書所建的許多檔案一樣，這個程式碼裡有一個物件叫 `game`，用來儲存遊戲有關的資訊。還有一個陣列叫 `baddies`（壞蛋）存在遊戲裡，裡面有遊戲裡每一個敵人的物件。這個函式迴圈壞蛋陣列，檢查每一個壞蛋物件，研判當時是否該執行更新程式碼。第 3 行給目前敵人設一個暫時參數 `ob`，這個目前敵人就是我們在壞蛋陣列裡正檢查的那個。下一行是增加 `ob` 裡的 `time` 屬性，由於判斷當時是否該執行更新程式碼的條件之一，是假如 `maxtime` 和 `time` 一樣，因此我們稍後（在第 11 行）執行這個檢查。第 5 和第 6 行是判斷敵人目前正在哪個方格上。而第 8 和第 9 行是判斷在影格最後敵人會在哪個方格上，這個方格被給了一個暫時參數叫 `cell_over_temp`。在第 11 行，我們檢查兩個條件判斷當時是否該執行更新程式碼，第一個條件是假設 `cell_over_temp` 不是空的（即裡面有物件），即執行更新程式碼。第二個條件是假設敵人物件上，`time` 變數值和 `maxtime` 變數值一樣，則也執行一次更新程式碼。

我們再來看看（從第 12 行開始的）更新程式碼。首先要把 `time` 歸 `0` 讓計時器重新開始，然後半隨機地設定一個新的 `maxtime` 值。這個隨機取樣的數值沒什麼特別，你可以改變這些數值，讓敵人做出不同的行為。如果你有興趣重新規劃這個 AI，並想控制它的難關，這一行程式碼可以玩玩看。接下來兩行（第 14 和第 15 行）是把敵人位置設定到它在影格開始時的位置。然後把敵人目前的方向存成 `tempDir`，就是「上」「下」「左」「右」四個字串值。然後再把 *x* 和 *y* 速率值（在第 17 和第 18 行）歸 0 以便重新計算。第 20 和第 21 行是求出 *x* 和 *y* 方向的正負號，判斷主角相對於敵人的位置。前面說過，*x* 和 *y* 的方向值可能是 -1 或 0 或 1。

第 22 到 24 行是加進了一個前面所說的「變笨」程序，它讓程式碼差不多每執行 10 次，就有一次把敵人的方向變成相反。

如果你有興趣依照小精靈的方式建立遊戲，這個 AI 可能很適合用於小精靈和那些鬼。

第 26 行判斷敵人是否與主角站在同一行或同一列，如果否，則隨機選擇 x 或 y 方向繼續前進。我們把不想去的方向設成 0，這樣當 xsign 設為 0 時，我們就在 y 方向走向主角。接下來第 34 到第 39 行，則根據哪個方向值為非 0，為 x 或 y 方向再做一次同樣的工作。我們也為非 0 的方向設定速度，並設一個變數 dir 儲存運動方向的字串值。這個字串值會在第五十二行用來顯示敵人影片片段內的某個影格，讓僵屍看起來像是走往正確的方向。第 53 和第 54 行是在敵人物件上儲存新建的 x 和 y 速率。

以上就是這個 AI。只要 AI 繼續被人使用，這個 AI 都會是最基本。但對簡單的遊戲而言，它已經夠用了。

11.3 完美的迷宮

許多人對迷宮遊戲都耳熟能詳，這一點相當有意思。因為好玩的猜迷遊戲會吸引人樂此不疲很長時間。這一節將討論建立隨機但完美的迷宮會用到的人工智慧。但首先你需要了解什麼叫做完美的迷宮。相對於不完美，一個完美的迷宮內，兩個方格間必有一條路徑，且任何兩格間只有一條路徑，繞圈的路徑不可能存在，因此沒有封閉的區域（見圖 11.3）。

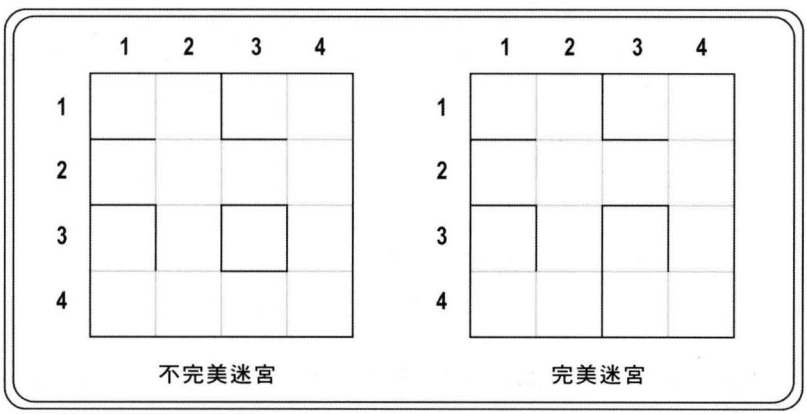


圖 11.3
不完美的迷宮和完美的迷宮。

圖 11.3 上可同時看到完美和不完美的迷宮。一個不完美的迷宮裡，會有封閉的方格，也會有某幾個方格中間有多條路徑。但一個完美的迷宮內，既沒有封閉的區域，且任兩個方格間只有一條路徑。圖 11.4 顯示的是一個更大且更好玩的完美迷宮。

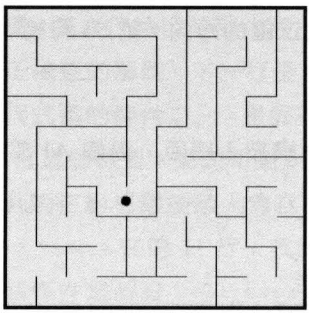


圖 11.4
一個更大的完美迷宮。

11.3.1 完美迷宮的規則

這是一個用二度空間陣列儲存每一個區塊資訊相當簡單的應用。迷宮是區塊世界，且每一區塊都有四堵牆。以下就是建立一個完美迷宮的規則。

1. 先決定迷宮要有多少行多少列，得出迷宮裡有「行×列」的方格數。這些方格目前都有每一面牆。然後再建立變數 `cellsVisited` 並設其值為 0。
2. 建立一個陣列叫 `visitedList`，當某個方格被到訪過，即被加進這個陣列。
3. 隨機選一個開始的方格做為目前的方格，並把 `cellsVisited` 的值增加。
4. 當 `cellsVisited` 的值等於區塊總數時，迷宮即已完成。否則就繼續到第五步。
5. 建立一個叫做 `neighbors` 的陣列，並檢視這個方格緊鄰的四個方格，分別稱它們為「東」「南」「西」「北」。再把任何未曾被到訪過的隔鄰方格加進 `neighbors` 陣列，如果任何隔鄰方格曾經被到訪過，即不把它們加進這個陣列。

6. 從 neighbors 陣列隨機選一個隔鄰方格。如果 neighbors 陣列是空的（表示所有的隔鄰方格都已經被到訪過），即前進到第九步，否則繼續到第七步。
7. 移動到這個隨機選擇的隔鄰方格上，打掉目前方格與這個隔鄰方格間的牆。
8. 讓這個隔鄰方格成為目前方格，並把它加進 visitedList 陣列。再回到第五步。
9. 移動到 visitedList 陣列內的前一個方格上，再從 visitedList 上刪除你目前所在的方格。回到第五步。

圖 11.5 所示是如何建立一個 3×3 迷宮的範例。

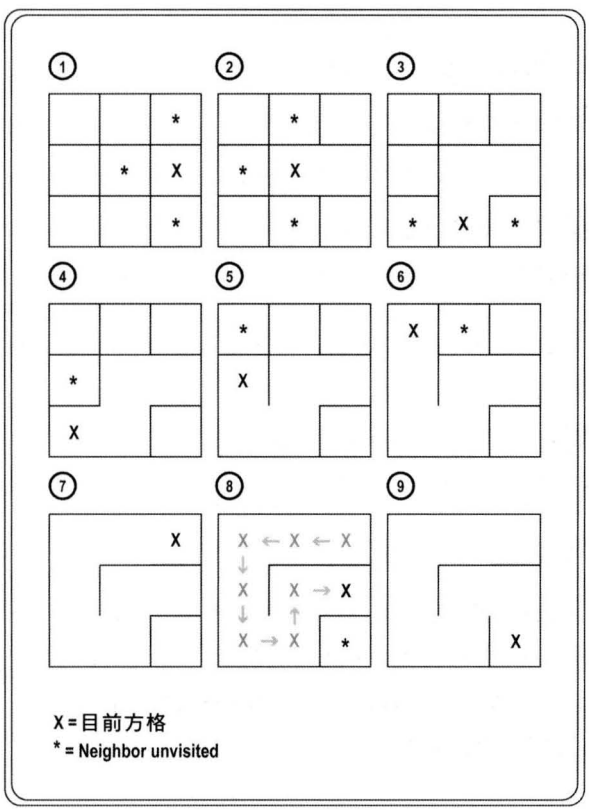


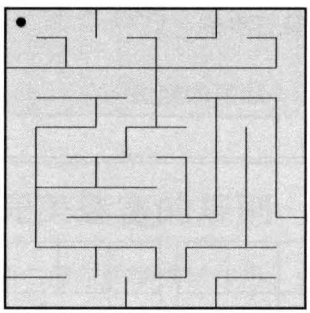
圖 11.5
建立迷宮。

11.3.2 使用 ActionScript 建立完美迷宮

這一節要探討的是如何用 ActionScript 編寫人工智慧運算法，並提供實例做為參考。

請打開光碟 Chapter11 的 maze.fla 檔。先執行 Test Movie 命令，看一下它的 SWF 檔。如果你初始化這個 SWF 檔幾次，應會發現迷宮每次都不一樣—但每次都必然完美且獨特。另外，無論哪一次，迷宮的左上角方格內一定都有一個點（見圖 11.6），這個點可以用鍵盤上的方向鍵移動走迷宮。

圖 11.6
左上角方格的點。



在 maze.fla 檔內一共有三個圖層：物件定義（Object Definition）、執行（Implementation）和素材（Assets）。物件定義圖層內有在記憶體裡建立迷宮的運算法，執行圖層內有在記憶體裡建立迷宮視覺圖像所需的 ActionScript，素材圖層內有顯示迷宮所需的影片片段。

我們最關心的是物件定義圖層裡的 ActionScript，因為它裡面有建立迷宮的人工智慧運算法。這個影格內的程式碼共有 75 行，都屬於一個長長的函式。

```
1     var maze:Object = new Object();
2     maze.createMaze = function(horizontal, vertical) {
3          this.rows = horizontal;
4          this.columns = vertical;
5          this.totalCells = this.rows*this.columns;
6          this.startRow = random(this.rows)+1;
7          this.startColumn = random(this.columns)+1;
8          this.cellsVisited = 0;
```

```
9          this.currentCell =
           ➔"cell"+this.startRow+"_"+this.startColumn;
10         this[this.currentCell] = {name:this.currentCell,
           ➔x:this.startRow, y:this.startComumn, exists:true};
11         this.visitList = [];
12         this.visitList.push(this[currentCell]);
13         while (this.cellVisited<this.totalCells) {
14              var cell:Object = this[this.currentCell];
15              var neighbors:Array = new Array();
16              if (cell.x-1>0) {
17                   //西面方格
18                   var x:Number = cell.x-1;
19                   var y:Number = cell.y;
20                   var westCell:String = "cell"+x+"_"+y;
21                   if (!this[westCell].exists) {
22                        neighbors.push([westCell, "west",
                          ➔"east", x, y]);
23                   }
24              }
25              if (cell.y-1>0) {
26                   //北面方格
27                   var x:Number = cell.x;
28                   var y:Number = cell.y-1;
29                   var northCell:String = "cell"+x+"_"+y;
30                   if (!this[northCell].exists) {
31                        neighbors.push([northCell, "north",
                          ➔"south", x, y]);
32                   }
33              }
34              if (cell.x+1<=this.rows) {
35                   //東面方格
36                   var x:Number = cell.x+1;
37                   var y:Number = cell.y;
38                   var eastCell:String = "cell"+x+"_"+y;
39                   if (!this[eastCell].exists) {
40                        neighbors.push([eastCell, "east",
                          ➔"west", x, y]);
41                   }
42              }
43              if (cell.y+1<=this.columns) {
44                   //南面方格
45                   var x:Number = cell.x;
```

```
46                      var y:Number = cell.y+1;
47                      var southCell:String = "cell"+x+"_"+y;
48                      if (!this[southCell].exists) {
49                          neighbors.push([southCell, "south",
                            ➔"north", x, y]);
50                      }
51                  }
52                  //隨機選擇一個隔鄰方格
53                  if (neighbors.length>0) {
54                      var nextCell:Number = random(neighbors.length);
55                      //打掉牆
56                      cell[neighbors[nextCell][1]] = true;
57                      //檢索新方格的名稱
58                      var newName:String = neighbors[nextCell][0];
59                      this[newName] = new Object();
60                      var newCell:Object = this[newName];
61                      newCell.exists = true;
62                      newCell.x = neighbors[nextCell][3];
63                      newCell.y = neighbors[nextCell][4];
64                      newCell.name = this.currentCell;
65                      //打掉牆
66                      newCell[neighbors[nextCell][2]] = true;
67                      this.currentCell = newName;
68                      this.visitList.push(this.currentCell);
69                      ++this.cellsVisited;
70                  } else {
71                      //走回上一方格
72                      this.currentCell = this.visitList.pop();
73                  }
74          }
75      };
```

函式一開始是建立一個物件 maze，然後給這個物件一個方法 createMaze()，這個方法接受兩個參數，指定迷宮要用來計算方格數的行列數，它們分別被存成 columns 和 rows。而這個迷宮內的方格總數即是行乘以列算出的積，這個值被存成 totalCells。然後（在第 7 行到第 9 行）隨機選一個方格做為起點，再把這個方格存成 currentCell（目前方格）。第 10 行則為這個起點方格建立一個物件做代表，並給物件四個屬性：name、x、y 和 exists（到訪）。其中，exists 的功用就是方便我們檢查方格有沒有被到訪過。若 exists 值為真，則方格曾被到訪

過。接下來是建立一陣列 visitedList，並把代表目前方格的物件放進這個陣列。現在，人工智慧有了起點，visitedList 陣列內也有一個到訪過的方格，接著我們可以執行 while 迴圈，直到 cellsVisited 變數等於 totalCells（第 13 行）。當 cellsVisited 等於 totalCells，迷宮即已完成。

第 14 行是給代表目前方格的物件建立一個參數。第 15 到 51 行是執行剛才所提到的第五步：建立 neighbors 陣列。然後我們就要檢查目前方格的西、北、東和南四個隔鄰方格，找找看哪一個方格還沒有被到訪過。如果找到一個，即把它加進 neighbors 陣列。如果我們決定要到訪這個方格，就把兩個方格內那面要打掉的牆的字串名稱存起來。比方說，如果這個方格是東面隔鄰方格，就把字串值「east」和「west」存起來。也就是說，如果選擇到訪這個方格，就要把目前方格的東牆打掉，並把東面方格的西牆打掉。雖然視覺上這個牆是同一堵牆，但在程式碼裡，每一個方格必須處理自己的牆。

第 53 行是開始進行第六步。假設 neighbors 陣列不是空的，則（在第 54 行）隨機選一個隔鄰方格。否則就進入 visitedList 陣列（第 70 到 73 行）。當隨機選擇了一個隔鄰方格後，我們就執行第七和第八步。同時必須做以下幾件事：

1. 為隔鄰方格建立一個物件。
2. 把目前方格和隔鄰方格間的牆打掉。
3. 增加 cellsVisited。
4. 把隔鄰方格設成目前方格。

以上這些事都在第 54 到第 69 行完成。如前所述，如果 neighbors 陣列內沒有元件，則我們就進入第 70 至 73 行，回到前一個方格。

11.3.3 完美迷宮的視覺效果

我不準備分析執行（Implementation）圖層的 ActionScript，但我要大致描述一下它在做些什麼事。首先，它會呼叫 maze.createMaze() 方法。在呼叫之後，maze 物件就會包含許多其他物件，名稱會像這樣：cell1_1、cell1_2、cell1_3

等等，和所有其他區塊世界遊戲的命名方式一樣。然後這個 ActionScript 就執行巢狀迴圈，把所有的區塊加到舞台上。在每一次重複時，ActionScript 會檢查 maze 物件裡的對應方格物件，並檢查它的 east 和 south 屬性。如果 east 值非真，則東牆就看得到。如果 south 值非真，則南牆也看得到。程式碼只管這兩個方格的東牆和南牆，因為 cell1_1 的東牆就是 cell2_1 的西牆，cell1_1 的南牆就是 cell1_2 的北牆。因此只需要這兩堵牆的資訊顯示這個牆一次，即可建立迷宮。

影格內其他的 ActionScript 是負責處理使用者走迷宮的影片片段。

11.4 尋找路徑運算法

就像我前面所說的，尋找路徑運算法就是尋找兩點間的任何路徑。而這兩點通常是指區塊世界裡兩個不同區塊的中心點（事實上，我從來沒有看過任何非區塊世界的尋找路徑）。以下是幾個最常見的運算類型。（註：這些運算都是在記憶體裡瞬間執行尋找路徑，然後通常以陣列形式顯示完整的路徑）

- 從第一個區塊開始，並在區塊間隨意走動，直到到達目標。
- 同時從開始區塊和目標區塊開始，隨意走動，直到路徑交叉。
- 從特定的起點開始向目標方向移動，碰到障礙物就沿著障礙物移動，直到繞過障礙物。這個尋找路徑的技巧叫做 tracing。許多現實生活裡的機器人也用這種技巧尋找路徑。

以上這幾種尋找路徑運算法各有優點也各有缺點，有的是運算超快但卻產出非常長而且看起來很怪的路徑，有的會找出看起來很不錯的路徑但只限在某些情況下，像是在一個沒有凹陷障礙物如衣櫥等的環境裡。因此，要用哪一種必須權衡利弊得失。

不過，前面說過。最有名的尋找路徑運算法是 A*。假設符合某些條件（等一下會再討論），A* 保證可以找出兩點間最短的路徑。但是就像其他的運算法一樣，A* 也有它的缺點：它跑得非常慢，也是最耗 CPU 的尋找路徑運算法（或之一）。但它卻是遊戲裡最常用的尋找路徑運算法。像《暗黑破壞神》裡，玩家在

畫面上某處按一下，角色就會走到那個位置。如果途中有障礙物，角色就會繞過障礙物繼續前進。任何遊戲能有這種尋找路徑的能力，大概用的就是 A*。這一節就來介紹這個運算法。

圖 11.7 所顯示的是 A* 尋找路徑運算法一項基本運用。裡面是一個 20×20 的格線。白色方格是空方格，黑色方格裡有牆，左上角（A 點）的灰色方格是角色的起點，右下角（B 點）暗灰色方格是角色的終點。兩點之間那條亮灰色的路徑，就是用 A* 尋找路徑運算法算出的路徑。

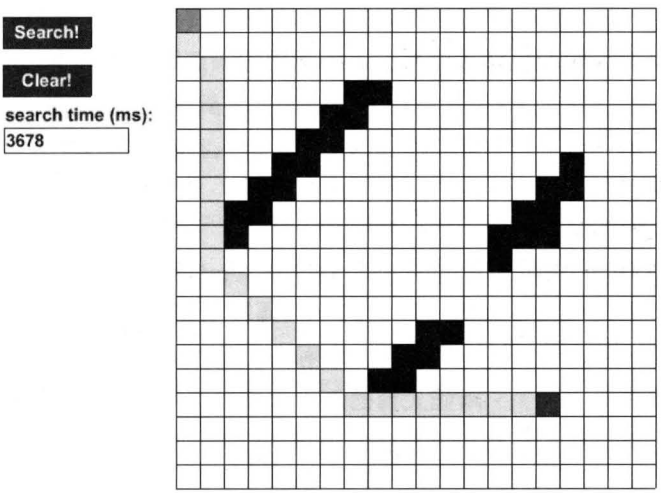


圖 11.7
A* 尋找路徑運算法找出 A 點到 B 點的路徑。

各種口味的 A*

線上遊戲是一項年產值達幾十億美元的產業，因此可想而知，許多人已投下大筆錢，開發更快更好的尋找路徑運算法，包括多種 A* 運算法的變體。這些 A* 變體通常都是 A* 運算法最佳化或修改之後的產品。本節最後將舉例介紹其中一種把速度最佳化的 A* 變體運算法。

11.4.1 A* 運算法

在介紹這個運算法之前，你必須先知道如何進行運算以及一些可用的工具。我已經把運算法寫成 ActionScript，光碟裡還有兩個範例都是用這個 ActionScript 建檔。可是我不打算一行一行解釋這個 ActionScript，反之，我準備說明如何使用我所建的檔，用模擬程式碼討論運算法的細節，以及這個 ActionScript 的一般應用。模擬程式碼是一種類似程式碼型式的運算（就像一章的概要），它談的是如何用程式碼工作，而不是提供特定的語法。這就是說，模擬程式碼不受什麼程式語言限制，Java 的程式設計師也好，ActionScript 的程式設計師也好，C++ 的也一樣，任何程式語言的設計師，都能看得懂。而模擬程式碼還有一點很棒，就是它一般都很短。比方說，這裡用的模擬程式碼只有 30 多行，但 ActionScript 寫的運算法將近 170 行。而模擬程式碼告知讀者從一個陣列刪除一個元件可能只有一行，但用真的程式碼，就必須先迴圈陣列找出元件再刪掉，就至少需要好幾行。

我的 A* 運算法是學自 www.gamasutra.com 網站（請見附錄 A「開發者資源」）找到的模擬程式碼，而我的 A* 運算法模擬程式碼雖然跟那個模擬程式碼很相像，但它們並不一樣。這就像兩個人同時坐下來寫一部電影如星戰二部曲的情節概要（本事），兩個人寫的內容大概差不了多少，但是描述方式多半各有千秋。這兩個模擬程式碼就是類似這樣的狀況。

不只用來尋找路徑

A* 運算法雖然主要是用來尋找路徑，但是它適用的範圍實際上更廣，可以用來解決許多類型的難題。我對 A* 用於尋找路徑知之甚詳，但對 A* 用在其他方面就一無所知。因此如果你有興趣在尋找路徑以外的其他方面使用 A* 運算法，可能上網找找會有不少資源可用。

A* 基本術語和功能

就像許多以數學為基礎的概念一樣，A* 尋找路徑運算法也用到許多術語，你必須先熟悉才能繼續學下去。這些術語主要是描述運算過程裡的位置、動作和結果：

- 節點（*node*）是指系統目前的位置。在尋找路徑裡，所謂目前位置就是指目前檢視的區塊，因此就我們所談的，一個節點就是一個區塊。
- 所謂「擴展」（*expand*）一個節點，這個動作（在程式碼裡）指的是到訪節點的每一個隔鄰節點。
- 一個推斷值（*heuristic*）在 A* 裡是指一個有憑有據的估計，根據一套精選的度量單位推估，得出的結果是一個數（等一下我們就會談到推斷值）。
- 「成本」（*cost*）是一個數量，是指我們給兩節點間移動指定的數值量。
- 「積分」（*score*）是成本及沿途到訪的每一個節點到目前節點的推斷值之總和。

了解這些術語之後，我們現在就來看看這些術語如何應用在運算法裡，或者應該說，這就來看看 A* 如何運作。前面說過，A* 會找出兩點間最短的路徑。但我們用的是什麼度量單位？時間？距離？還是步數？基本上 A* 適用於任何一種度量單位，不過我們選用的是距離。其他會用到的度量單位可能是時間（求最短時間內可走到的路徑）或是用了幾箱汽油（求汽車用最少量汽油可到的路徑）。我們指定一個區塊與另一個水平或垂直隔鄰區塊間移動的成本（cost）值為 1（一英呎、一英浬或一個區塊都無所謂），而一個區塊與另一個斜角隔鄰區塊間移動的成本值為 1.41。1.41 是兩個隔鄰斜角區塊中心點間的距離。

推斷值（heuristic）是對目標區塊中心點與（尋找路徑時正在檢視的）目前區塊中心點間距離的最佳猜測。做這個猜測相當容易，只需要簡單的邏輯。公式就是，每當被到訪後，每一個節點會被指定一個積分 f：

$$f = g + h$$

h 就是推斷值，即該區塊與目標區塊間距離的最佳推測。g 是每一個沿途到訪過的節點到目前節點之積分總和。這個概念用對比的方式可能最容易理解。比方說計劃從紐約到巴黎旅行，但預算很緊，於是就想找一條最省錢的路徑。現在把紐約到巴黎間沿途的城市例如紐約、倫敦、里斯本、布魯塞爾、馬德里、巴黎都視為節點。在研究成本的過程中，計算過紐約到倫敦的成本後先存起來，同樣的也把里斯本到紐約、倫敦到巴黎等的旅行成本一一計算後儲存起來。到最後，如果應用 A* 其他的規則（還沒有討論過），即可找出（以成本而言）最佳路徑。假設這條最佳路徑最後是紐約—馬德里—倫敦—巴黎。在紐約（如同在其他節點）$f=g+h$。g 是前面所有節點之 f 的總和。由於紐約是起點，沒有上一層（parents）節點，因此在紐約的 $g=0$。對馬德里而言，雖然也是 $f=g+h$（如在所有的節點）。但在這個情況下，g 不是 0，因為這時是從另一個節點到訪馬德里。這裡的 g 值就包含從紐約出發的 f，因此 g 是到目前節點成本的連續總和。如果你真的旅行這一趟，那麼 g 就是一直到目前地點所花的費用。

現在我要介紹 A* 最了不起的功能之一—它處理地形的方式。前面說過，一個區塊移動到另一個區塊的成本可能是 1 或是 1.41。如果所有區塊的大小都一樣，這個說法正確。但這個陳述並不一定永遠都為真。比方說某些區塊是水時，你很可能除非絕對必要不會讓角色過水。因此你就指定一個成本如 10 給任何有水的節點間移動（從一個節點移到另一個節點）。雖然這樣並不保證路徑一定不會經過水，但它會最優先選擇不經過水的路徑。而若水是一條穿過整個地圖的一條溪，而且中間沒有橋，那 A* 最後一定會給一條穿過水的路徑。但若中間有橋，而且離得很近，那 A* 就會給一條經過橋的路徑。另類情況下，如果角色是半人半魚，那 A* 可能寧願選擇過水。在這種情況下，陸地的成本就可能低於水。綜合這裡所討論的，我現在可能要修正我前面所說 A* 永遠都會找出最短路徑的說法。因為你現在懂得比較多，我可以更進一步界定：A* 永遠會找出積分最低的路徑。而在許多情況下（像某些地形沒有改變的遊戲世界，如這一節稍後用到的），積分最低的路徑剛好都是最短的路徑。

A* 說的是英語

這裡就來看模擬程式碼敘述的尋找路徑運算法。

```
1     AStar.Search
2          create open array
3          create closed array
4          s.g = 0
5          s.h = findHeuristic( s.x, s.y )
6          s.f = s.g + s.h
7          s.parent = null
8          push s into open array
9          set keepSearching to true
10         while keepSearching
11              pop node n from open
12              if n is the goal node
13                   build path from start to finish
14                   set keepSearching to false
15              for each neighbor m of n
16                   newg = n.g + cost( n, newx, newy )
17                   if m has not been visited
18                        m.g = n.f
19                        m.h = findHeuristic( newx, newy )
20                        m.f = m.g + m.h
21                        m.parent = n
22                        add it to the open array
23                        sort the open array
24                   else
25                        if newg < m.g
26                             m.parent = n
27                             m.g = newg
28                             m.f = m.g + m.h
29                             sort to open array
30                             if m is in closed
31                                  remove it from closed
32              push n into the closed array
33              if search time > max time
34                   set keepSearching to false
35         return path
```

這個運算法用到了兩個表（list）（即 Flash 裡的陣列）：open 和 closed。其中 open 陣列裡是曾經到訪過的節點，closed 陣列裡是所有擴展過的節點（即這些節點的所有隔鄰都已經到訪過）。我們用 open 陣列做為優先權排列陣列（priority queue），它不僅儲存節點，而且是以一種順序儲存節點，就是從最低積分（f）排到最高積分。而每當在 open 陣列裡加進一個節點，或在 open 陣列改變一個節點的 g 值時，都要重新排序一次，讓節點依積分從最低排到最高。

第 2 和第 3 行是建立一個空的 open 和 closed 陣列。由於尋找路徑需要一個起點和一個終點，接下來就是設定起點和終點。s 是一個物件，代表開始節點，設 s.g 為 0 是因為起點節點沒有上一層（parent），因此（第 4 行）到達 s 的成本（g）就是 0。再來是為起點節點找出推斷值 h（推斷值是指從目前節點到目標節點估計所需成本）。然後（在第 6 行）把起點節點上 s.g 加 s.h 之和存成 f 值。由於 s 沒有上一層（parent），因此這裡設 s.parent 為 null（零）。下一步（第 8 行）是把 s 節點推進 open 陣列，成為 open 陣列裡第一且唯一的節點。

第 9 行設定變數 keepSearching 值為真，只要它一直為真，程式就一直執行 A* 尋找路徑。但當程式判斷找到一條路徑時，或沒有路徑存在時，或找路的時間過長時，keepSearching 就被設為偽。

第 11 行是從優先權排列陣列裡選出一個節點，檢查它是不是終點節點。如果它是，則已到達終點，即（在第 12 行到第 14 行）停止找路，建立路徑。如果它不是目標節點，則擴展該節點，也就是說，到訪這個節點的每一個隔鄰節點。第 16 行是求出目前檢查的隔鄰節點 m 的 g 值。然後檢查這個節點是否已曾到訪過，如果否，則把運算法填進第 18 到 23 行。這裡設定 m 上 g 的值，它就是它上一層 n 的 f 值。接著我們要計算並儲存推斷值和 m 上的 f 值。最後設定上一層屬性為前一節點 n 的屬性。如果這個節點以前到訪過，而現在有一個較低的 g 值，則把運算法填進第第 25 到 31 行。

這裡我要對 g 詳加說明。當一個節點第一次被到訪過以後，程式會根據到達它的路徑給它指定一個 g 值（就像我前面所說的）。但在尋找另一條可能的路徑時，那個節點非常可能被再度到訪。而如果新路徑的 g 值低於前面已存路徑的 g 值時，就（在第 27 行）用新的 g 取代舊的 g。第 26 行設定 m 的上一層（parent）

屬性為來自的（譯註：即前一個）節點。我們會在尋找的最後用上一層屬性建構最後路徑，也就是說，程式可以循著上一層屬性從目標節點一路走回起點節點。接下來是再計算一次 *f*，然後必須把 open 陣列重新排序，因為我們剛剛以較低的 f 值更新其中一個節點，使這個節點可能取得了比其他節點高的優先權。

有一點我必須承認，我不是很清楚第 30 和第 31 行是做什麼用的。它們在我學 A* 運算法的那個模擬程式碼裡就有，所以我把它們放進我的 ActionScript 裡。我曾經建構多次尋找範例測試這個部份的運算法，但始終未曾碰到過（我是在那段程式碼裡放進一個 trace 命令，好讓我在碰到它時被告知）。而經過幾十次測試，卻始終未曾碰過用到它的時候。雖然未曾用過這段程式碼，ActionScript 寫的這個運算法就我所知一直都運作得很正常，給的一直都是最低積分的路徑。不過我還是把這段運算法保留，既使看起來似乎無此必要，但還是以備哪一天我突然了解什麼時候需要用到它。如果你對 A* 非常精，拜託請告訴我你對這段運算法的看法！

在 n 的所有隔鄰節點都被到訪過之後，就跳到第 32 行，把 *n* 推進 `closed` 陣列，因為這時它已完成擴展。然後再檢查尋找時間是不是用得太久，如果是，則把 `keepSearching` 設為偽。否則跳到（第 11 行）優先權排列的下一個節點。而若 `keepSearching` 的值為偽，則停止尋找並建立路徑。

在看過 A* 運算法之後，如果有看沒有懂，請先不要沮喪，因為它真的不是很容易懂。我也是在網上寫過好幾篇文章之後，才好像覺得完全懂了基本的 A*。

執行 A*

無論如何，在看過 A* 運算法，並基本了解它如何運作之後，接下來就看看我用 ActionScript 把 A* 運算法寫成的類別 `Astar`。

請打開光碟內 Chapter11 下 Astar 目錄裡的 astar.fla 檔，匯出 SWF 檔看看。裡面是一個等比例區塊世界，有一個角色站在一個角落，地面都是草地。

在 SWF 檔的右上角可以看到幾個選項按紐，上面一行選項按紐是用來改變這個世界裡個別區塊的地形，下面一行是用來把物件（如植物或磚塊）加到區塊裡。

現在請選擇並按下任一區塊。然後請看 SWF 檔的右下角，那裡有一個按紐叫 search（尋找）。當按下這個按紐，一條從角色所站區塊到剛按下那區塊間的路徑即被找出（見圖 11.8）。

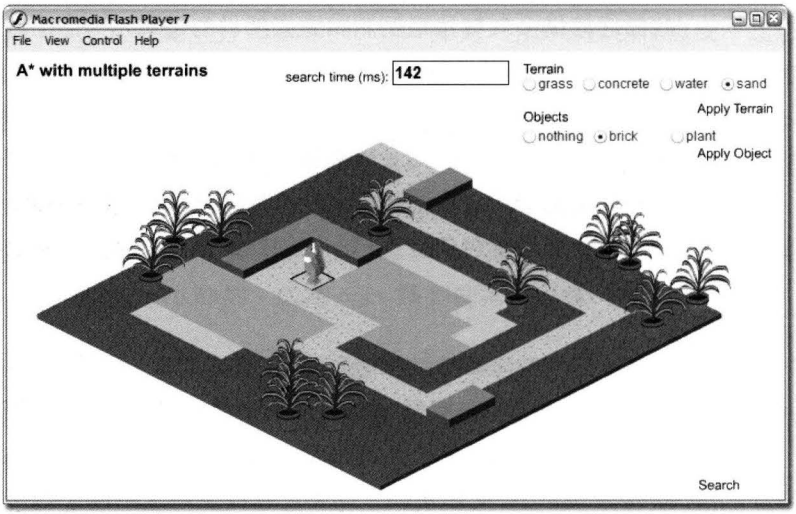


圖 11.8
Astar。

這個檔是測試 A* 在不同地形如何尋找路徑。你在實驗時會看到，角色是用最低成本走這條路徑。如果在世界裡建立一長條水道，同時指定角色走到水道另一邊的某個區塊上，這時角色就會繞過水道走到那個區塊。但若 A* 判斷繞過水道的成本太高，角色就會被安排走過水道。

這個檔案用了四個自訂類別：

- 地圖（Map）—使用地圖類別是為便於建立區塊世界。這個類別裡有方法（method）可用來取得區塊的資訊、把物件（如植物、鑰匙等）加進區塊裡，以及為區塊設定地形。

- Astar—這個類別裡有 A* 尋找運算法的邏輯，是用來尋找兩個節點間最低成本路徑。而當執行一次尋找時，一個地圖類別實體會被傳遞進 Astar 類別，讓 Astar 檢查區塊的地形，傳回的結果即為最低成本路徑。
- 優先權排列（PQueue）—這是一個優先權排列類別，會根據成本為一個節點的陣列排序，也提供有用的方法（method）從排列中新增或移除節點。
- 等比例（Isometric）—等比例類別就是在等比例世界觀那一章所介紹過的那個類別。

以下是這個檔裡所用的各個地圖類別方法：

- new Map(columns, rows)—建立新的地圖類別實體。當這個方法被呼叫時，是把地圖空間裡的行列數傳遞進去。這個檔裡所建的地圖行列數是 15×15。
- addTerrain(terrain_array)—這個方法是，在記憶體裡，在地圖上建立指定的地形。陣列內每一個元件都只是一個字串。
- setTransitionCost(terrain1, terrain2, cost, reversible)—從一個區塊移動到另一個區塊的相關成本，會隨著起始區塊的地形和最後區塊的地形改變。方法內前兩個參數是代表地形類型的字串，第三個是從 terrain1 到 terrain2 的成本值，第四個參數是一個 Boolean 值。若 reversible（反算）的值為真，則從 terrian1 到 terrain2 相關的成本，也會與從 terrain2 到 terrain1 的成本相關。若 reversible（反算）的值為偽，則成本即不指定給反向情況。比方說，從沙地移動到流沙很容易，但從流沙移到沙地就非常困難，因此反向成本必然不同。
- getTile(column, row)—把行與列的參數傳給區塊。
- setStart(column, row)—設定路徑起點，給 Astar 類別使用。
- setGoal(column, row)—設定路徑終點，Astar 類別也會用到。

以下是這個檔裡所用的各個 Astar 方法：

- new Astar()—這個方法是用來建立新的 Astar 類別實體。
- search(map)—這個方法是用來尋找兩個區塊間最低成本路徑。一個地圖類別實體的參數會被傳遞進去。而地圖類別實體的 setStart() 和 setGoal() 方法必須先呼叫，好讓 Astar 類別知道要用哪兩個區塊。這個方法是傳回一個物件做為結果。這個物件有以下屬性：
  - path—這是起點區塊到終點區塊路徑內的一個區塊陣列。
  - setMaxSearchTime(milliseconds)—Astar 類別尋找路徑的時間預設值為 1,700 毫秒（1.7 秒）。如果在這個時間內沒有找到路徑，它就會停止尋找。可用這個方法改變最大尋找時間。
  - pathFound—這是一個 Boolean 值。若為真，即找到一條路徑。若為偽，即沒有找到路徑。而找不到路徑，有可能是目的地區塊被無法通過的區塊所包圍，或者尋找時間超過 maxSearchTime 的值，即預設的 1,700 毫秒。
  - searchTime—這是尋找路徑所需的時間，單位是毫秒。

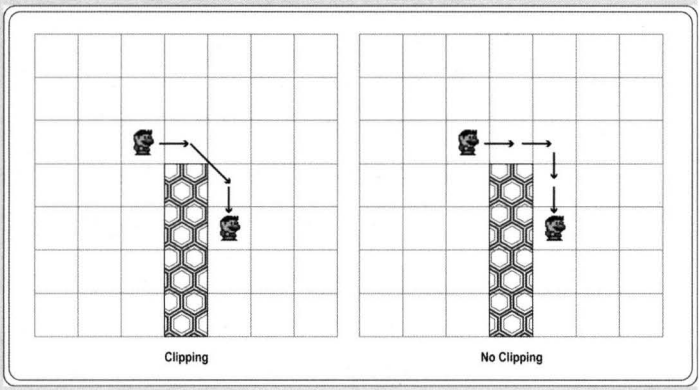
你不會直接接觸到 PQueue 類別，因為 Astar 類別是在內部用來為 open 表（陣列）排序。

遷就現實繞點路

Astar 類別內有一個屬性叫做 preventClipping（不走捷徑），如果它的值為偽，即會真的傳回一個最短路徑。在這種情況下，角色可以斜著從一個區塊走到別的區塊，而且在它移動過程中，兩個相鄰區塊會分別顯示角色的一半。但是有一個條件，就是隔鄰區塊間沒有障礙物，才會出現這種情況。而若隔鄰區塊間有障礙物，角色就會有部份顯示為穿過障礙物的一角。不過還好有辦法解決這種不夠真實的行為。由於 preventClipping 屬性的預設值為真，如果隔鄰區塊之一有障礙物，運算法就不會執行（return）區塊間斜向移動。

這會讓路徑看起來更接近真實，卻不一定是最短路徑。然而它一定是視覺逼真路徑中最短的路徑（見圖 11.9）。

圖 11.9
捷徑（clipping）與非捷徑（no clipping）顯示圖。



在這個 A* 執行過程裡，我修改了 `cost()` 函式，以讓正常情況下會走捷徑的斜向移動傳回非常大的值，結果由於 `preventClipping`（不走捷徑）屬性預設值為真，路徑就會較自然地繞過物件，而不會走捷徑。這時仍舊是斜向移動，但不是走捷徑那樣地斜向移動。而如果你比較喜歡使用純的 A* 路徑，就用 `setPreventClipping()` 方法把 `preventClipping` 值設為偽。這個方法的預設值為偽。

還覺得不夠快？

你應該已注意到，astar.fla 檔找到路徑的速度非常慢。而 iso_ai_astar.fla 檔內的速度只到可接受的程度。在那個檔案裡，尋找的時間會從 80 毫秒（ms）慢到 400 毫秒（至少我的電腦上會這樣）。如果你有興趣用 ActionScript 讓 A* 執行得如閃電一般快（但並不實用—因為沒有地形支援），請參閱 astar_optimized 資料夾，那裡面有一些 Casper Schuirink 所建非常快的 A* 檔。在我的 A* 裡要 200 毫秒的尋找，Casper 的 A* 可在 10 到 40 毫秒完成，而其大部份檔案的尋找所需時間都在 50 毫秒以下。但就像所有高度最佳化檔案一樣，它們也都有缺陷。在這裡主要的缺點是檔案並不一定容易放進你的應用程式裡，這是為速度犧牲掉使用的方便性，所以使用時請注意。

11.5 重點回顧

- 尋找路徑就是尋找一個點與另一個點間路徑的動作。
- 人工智慧可能被用到的兩個主要領域是尋找路徑和控制敵人行為。
- 人工智慧可用來隨機建立地圖。
- 使用特別的運算法可建立隨機但完美的迷宮。
- A*（唸為 A Satr）是目前所知最好的尋找路徑運算法。如果有路可走，它保證會找到兩點間最低積分的路徑。
- 雖然 A* 是最好的尋找路徑運算法，但它也很耗 CPU 且非常慢。
- A* 可以處理多種類型的地形，並找出經由這些地形到達目標的最佳路徑。