電腦對局導論

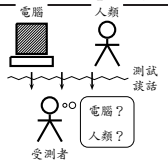
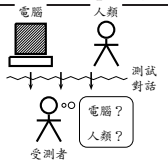
Computers and classical board games: An Introduction

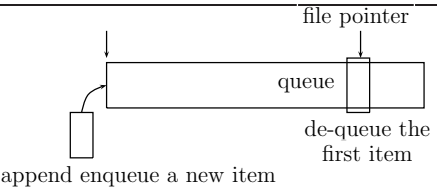
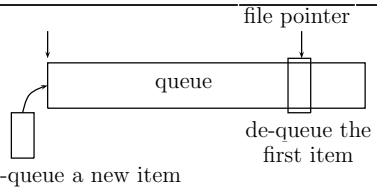
2017年 6月一版

勘誤表

August 23, 2021

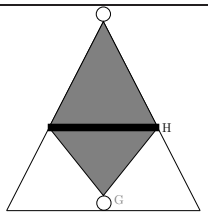
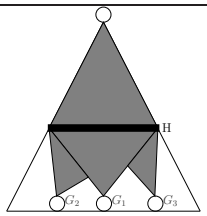
頁碼/位置	內容	
	修改前	修改後
p.I 序一的第二段	預官退伍之後，讚昇出國到美國德州奧斯汀大學深造，專攻演算法研究。	預官退伍之後，讚昇出國到美國德州大學奧斯汀校區深造，專攻演算法研究。
p.I 序一的第三段	2005年8月，讚昇與我共同主辦第十屆國際電腦奧林匹亞大賽和CG2005電腦對局國際會議，開啓ICGA國際電腦對局學會在亞洲地區舉辦活動的新頁。	2005年8月，讚昇與我共同主辦第十屆國際電腦奧林匹亞大賽和CG2005電腦對局國際會議，開啓ICGA國際電腦對局學會在亞洲地區舉辦活動的新頁。
圖目錄之 2.4	混合雙佇列實作佇列之示意	混合雙佇列實作佇列之示意圖
圖目錄之 4.4	六貫棋性質證明：連接黑方棋子	六貫棋性質證明：連接各行中的黑方棋子
圖目錄之 5.7	位在中央的騎士	騎士的影響
圖目錄之 5.14	栓鏈的範例	栓鏈
圖目錄之 5.19	欠行局例	欠行
圖目錄之 7.4	斥候演算法搜尋的節點數比 Alpha-Beta 切捨演算法拜訪的節點數多的例子	斥候搜尋時 TEST 拜訪的節點數比 Alpha-Beta 切捨多的例子
圖目錄之 7.8	斥候演算法拜訪最少的節點數的例子	斥候演算法拜訪最少節點數的例子
演算法目錄之15	15 F′(position *p*)	15 F′(position *p*, integer *depth*)
演算法目錄之16	16 G′(position *p*)	16 G′(position *p*, integer *depth*)
演算法目錄之17	17 F(position *p*)	17 F(position *p*, integer *depth*)
演算法目錄之18	18 F2′(position *p*, value *alpha*, value *beta*)	18 F1′(position *p*, value *alpha*, value *beta*)
演算法目錄之19	19 G2′(position *p*, value *alpha*, value *beta*)	19 G1′(position *p*, value *alpha*, value *beta*)
演算法目錄之20	20 F2(position *p*, value *alpha*, value *beta*)	20 F2(position *p*, value *alpha*, value *beta*, integer *depth*)
演算法目錄之21	21 F2(position *p*, value *alpha*, value *beta*)	21 F2(position *p*, value *alpha*, value *beta*, integer *depth*)
演算法目錄之29	29 IDAS(position *p*, integer *limit*, integer *threshold*)	29 IDAS(position *p*, integer *limit*, value *threshold*)
演算法目錄之30	30 IDAS′(position *p*, integer *limit*, integer *threshold*)	30 IDAS′(position *p*, integer *limit*, value *threshold*)
演算法目錄之36	36 F4.4′(position *p*, value *alpha*, value *beta*, integer *depth*, Boolean *do_null*)	36 F4.4′(position *p*, value *alpha*, value *beta*, integer *depth*, Boolean *in_null*)
演算法目錄之37	37 F4.5′(position *p*, value *alpha*, value *beta*, integer *depth*, Boolean *do_lmr*)	37 F4.5′(position *p*, value *alpha*, value *beta*, integer *depth*, Boolean *in_lmr*)
演算法目錄之45	45 UCT	45 MCTS
演算法目錄之59	59 F2.1′(position *p*, value *alpha*, value *beta*)	59 F3.1′(position *p*, value *alpha*, value *beta*)

演算法目錄之60	60 Star1_F2.1′(position p, node n, value $alpha$, value $beta$)	60 Star1_F3.1′(position p, node n, value $alpha$, value $beta$)
p.3 的圖 1.1		
p.7 的 1.3 節	一千年前就有這種非常好的演算法分析，而且詳細記載並分析了使用自我連續相乘的技巧來快速計算 3^n 的演算法。	一千年前就有這種非常好的演算法分析，而且詳細記載並分析了使用自我連續相乘的技巧，不管 n 是不是 2 的次方，都可以快速計算 3^n 的演算法。
p.7 的 1.4 節	Barbara Liskov 2008年）	Barbara Liskov（2008年）
p.9 的 1.4.4 節	因爲西洋棋被認爲是相當複雜且具有高度智慧的行爲表現，1997年IBM的深藍超級電腦（Deep Blue Supercomputer）打敗人類的西洋棋棋王Garry Kasparov被視爲電腦弈棋研究中最爲重要里程碑之一。	因爲西洋棋被認爲是相當複雜且具有高度智慧的行爲表現，1997年IBM的深藍超級電腦（Deep Blue Supercomputer）打敗人類的西洋棋棋王Garry Kasparov被視爲電腦弈棋研究中最重要的里程碑之一。
p.10 的 1.4.4 節	· · · 揭開電腦圍棋對局程式研究的新頁。	· · · 揭開電腦圍棋對局程式研究的新頁，2018 年後更發展出不需監督之深度學習軟體 Alpha Go Zero 及 Alpha Zero 成果卓然。
p.11 的 1.5.2 節	目前觀察到一個謎題謎題必須至少要有 NP 完備的難度才會耐玩。	目前觀察到一個謎題必須至少要有 NP 完備的難度才會耐玩。
p.12 的 1.5.3 節	以圖 1.3 爲例，研究學者透過 KINECT 和三維井字遊戲（Qubic）程式進行互動。	以圖 1.3 爲例，研究學者透過微軟公司之 KINECT 裝置和三維井字遊戲（Qubic）程式進行互動。
p.16 的 2.1.1 節	開發者依照這些特徵程式自行作權衡，找出最有利於程式開發的搜尋演算法。	開發者依照這些程式特徵自行作權衡，找出最有利於程式開發的搜尋演算法。
p.18 的 2.2 節	分成「單純型」（pure）暴力搜尋演算法和「智慧型」（intelligent）暴力搜尋程式；前者有可能會重複拜訪某個節點多次，也不保證所有節點都會被拜訪過，後者針對前者之缺點加以修改，使拜訪任一節點的次數不超過某個上限次數。	分成「單純型」（pure）暴力搜尋演算法和「智慧型」（intelligent）暴力搜尋演算法；前者有可能會重複拜訪某個節點多次，也不保證所有節點都會被拜訪過，後者針對前者之缺點加以修改，使拜訪任一節點的次數不超過某個上限次數。
p.18 的 2.2 節	Algorithm 1 第 5 行若是採用隨機選取下個節點的方式，而不是隨意選取下一步的進行方式，我們稱之爲隨機走步法（random walk）。	Algorithm 1 第 5 行若是改用挑取一個隨機節點，而不是現行的任意節點，我們稱之爲隨機走步法（random walk）。
p.18 的 2.2 節	故智慧型暴力搜尋演算法在建立一個有效率地拜訪節點的策略下，能夠拜訪所有節點。	故智慧型暴力搜尋演算法旨在建立一個有效率地拜訪節點的策略，用以拜訪所有節點。
p.18 的 2.2 節	故此，在善用硬體計算資源的前提下，投入智慧型暴力搜尋演算法的研究是相當有回報的，接下來介紹數個較著的名智慧型暴力搜尋演算法。	故此，在善用硬體計算資源的前提下，投入智慧型暴力搜尋演算法的研究是相當有回報的，接下來介紹數個較著名的智慧型暴力搜尋演算法。
p.21 的 2.2.1 節	用一個聰明的策略來確定已處理列表內的節點被重複拜訪的次數不會太多次，且必須追蹤目前訪問到的所有節點，假設此拜訪的節點深度爲 d，探討一次此深度的節點則需花費的時間複雜度爲：· · ·	用一個聰明的策略來確定已處理列表內的節點不會被重複拜訪多次，且必須追蹤目前訪問到的所有節點，假設此拜訪的節點深度爲 d，探討一次此深度的節點則需花費的時間複雜度爲：· · ·

p.21 的 2.2.1 節	儘管一定找得到最佳解的優點，但卻有個嚴重的缺點就是需要大量的儲存空間。	儘管一定找得到最佳解，但卻有個嚴重的缺點就是需要大量的儲存空間。
p.21 的 2.2.1 節	根據此缺點，可使用硬碟代替記憶體，以改進儲存空間不足的問題，一般而言，雖可勉強忍受增加10倍的計算時間，卻不能忍受增加10倍量的硬體資源，因爲硬體需求增加太多的話，作業系統通常無法處理，會造成系統完全失效。	根據此缺點，可使用硬碟代替記憶體，以改進儲存空間不足的問題，一般而言，增加10倍的計算時間可勉強忍受，卻不能忍受增加10倍量的硬體資源，因爲硬體需求增加太多的話，作業系統通常無法處理，會造成系統完全失效。
p.22 的 2.2.3 節	其概念如下述：將需儲存佇列內的資料進行排序後存入硬碟內，並且隨時保持硬碟內資料在任何時候都排序好的狀態。硬碟爲儲存佇列內容的位置，空下的記憶體則可記錄最近被探尋過的節點資料，此舉可有效利用計算機之硬體空間，但記憶體空間依舊有限，倘若記憶體空間已滿而需進行記憶體的資料清理，將較早探尋的資料刪除，以期有效利用空間。	其概念如下述：將需儲存在佇列內的資料進行排序後存入硬碟內，並且隨時保持硬碟內資料在任何時候都是排序好的狀態。硬碟爲儲存佇列的內容，所空下的記憶體可記錄最近被探尋過的節點資料，此舉可有效利用計算機之硬體空間，但記憶體空間依舊有限，倘若記憶體空間已滿而需進行記憶體的資料清理，將較早探尋的資料刪除，以期有效利用空間。
p.22 的 2.2.3 節	以圖 2.3 爲例，可用系統函式$lseek()$尋找目前磁碟佇列中檔案佇列的開頭。	以圖 2.3 爲例，可用系統函式$lseek()$（使用法請見相關作業系統教科書）尋找目前磁碟佇列中檔案佇列的開頭。
p.22 的圖 2.3		
p.24 的圖 2.4 之標題	混合雙佇列實作佇列之示意	混合雙佇列實作佇列之示意圖
p.24 的 2.2.3 節	*current* 是 $deeper(N)$ 的某一個 N 的子節點$next(current, N)$ 會依序回傳在 $deeper(N)$ 的次序中 *current* 的下一個 N 的子節點。	*current* 是 $deeper(N)$ 的某一個 N 的子節點，$next(current, N)$ 會依序回傳在 $deeper(N)$ 的次序中 *current* 的下一個 N 的子節點。
p.24 的 2.2.3 節	深度優先搜尋演算法因需要記錄正在拜訪的路徑（由起始節點開始依序記錄）於堆疊內，故在深度爲 d 的搜尋樹上，廣度優先搜尋的空間複雜度爲 $O(d)$。如同討論廣度優先搜尋的時間複雜度，歪斜樹存在著一個最差的空間複雜度爲 $O(D)$。通常 D 不太大，所以可以放在記憶體中，因此不需要使用硬碟協助。若 D 實在太大，目前並無好的硬碟演算法。	深度優先搜尋演算法因需要記錄正在拜訪的路徑（由起始節點開始依序記錄）於堆疊內，故在目標深度爲 d 的搜尋樹上，深度優先搜尋的空間複雜度爲$O(d)$。此外在歪斜樹上空間複雜度最差可能爲 $O(D)$。通常 D 不太大，所以可以放在記憶體中，因此不需要使用硬碟協助。若 D 實在太大，目前並無好的硬碟演算法可解決。
p.25 的末段	深度優先搜尋演算法可能因爲缺乏好的切捨（pruning）深度，而不能快速找到目標，故使用此搜尋法不能保證能夠獲得最佳解。	深度優先搜尋演算法可能因爲缺乏好的切捨（pruning）深度，而不能有效率地找到目標，此外使用此搜尋法並不能保證能夠獲得最佳解。
p.26 的首行	...也就是如何在數個著手時做出的抉擇中，...	...也就是如何在數個著手時做出的抉擇，...

p.26 的 2.2.3 節之末段	針對此缺失，可利用雜湊表（hash table）記錄已被搜尋過的節點之方式來解決，但需顧慮到雜湊表儲存在硬碟中是否會因過度讀取導致時間效能的低落，必要時以資料壓縮技術或位元運算（bitwise operation）管理重複的節點會是一個努力的方向。相關的著手知識設計原則及雜湊表技巧，將在第 8 章中介紹。	針對此缺失，可利用雜湊表（hash table）記錄已被搜尋過的節點之方式來解決，但需顧慮到雜湊表儲存在硬碟中是否會因過度讀取導致時間效能的低落，可以資料壓縮技術或位元運算（bitwise operation）管理重複的節點來降低衝擊，會是一個努力的方向。相關的著手知識設計及雜湊表技巧，將在第 8 章中介紹。

p.27 的 Algorithm 5	13 **for each state** Z in $deeper(N)$ **do**
	14 **if** Z is not in P **then**
	15 $Push(S, Z)$; // to avoid loops
	16 $Push(S, null)$; // marker for end of search siblings
		13 $Push(S, null)$; // marker for end of search siblings
		14 **for each state** Z in $deeper(N)$ **do**
		15 **if** Z is not in P **then**
		16 $Push(S, Z)$; // to avoid loops

p.29 的 2.2.6 節	漸進式深度優先搜尋演算法承襲了廣度優先搜尋的省記憶體空間之優點，在目標節點爲 d 空間複雜度爲 O(d)。	漸進式深度優先搜尋演算法承襲了深度優先搜尋的省記憶體空間之優點，在目標節點深度爲 d 之空間複雜度爲 O(d)。
p.30 的 2.2.6 節	深度優先搜尋與逐層加深深度優先搜尋最大的不同點是，前者不需要考慮剪枝的問題，而後者則需要考慮剪枝，故好的剪枝對於逐層加深深度優先搜尋演算法是重要的。此外，需建立一個機制來確某一節點是否已經被拜訪過。	深度優先搜尋與逐層加深深度優先搜尋最大的不同點是，前者需要考慮剪枝的問題，而後者則不需要考慮剪枝，故設計好的切捨深度對於沒有逐層加深深度優先搜尋演算法非常重要。此外，逐層加深版也需建立一個機制來確定某一節點是否已經被拜訪過。
p.31 的 2.2.7 節	不論是廣度優先搜尋或者是深度優先搜尋都沒有方向性的探討，若有方向性的探討，可以針對節點狀態來探討，探討能搜尋到此節點狀態的後向狀態、以此節點狀態可以搜尋到的節點狀態有哪些，增加此方向性的運算，使深度優先搜尋更具發展性。	之前的深度優先搜尋都沒有方向性的探討，若有方向性的探討，可以針對節點來研究，探討能搜尋到此節點的後向節點、以此節點可以搜尋到的前向節點有哪些，增加此方向性的運算，使深度優先搜尋更具發展性。
p.32 的圖 2.6		
p.34 的 2.3.1 節末段	在將路徑加入 PQ 的時候，需要檢查這個路徑是否到達其他之前已經拜訪過的節點，這個動作需要耗費大量的記憶體空間和完備的演算法來實作有效率的優先權佇列資料結構，且需要好的資料結構和演算法來檢查路徑是否重複。	在將路徑加入 PQ 的時候，需要檢查這個路徑是否到達其他之前已經拜訪過的節點（第 9 至 13 行），這個動作需要耗費大量的記憶體空間和完備的演算法來實作有效率的優先權佇列資料結構，且需要好的資料結構和演算法來檢查路徑是否重複。
p.37 的 2.3.4 節末段	IDA*演算法擁有成本花費知識，結合上述搜尋的演算法， 若每次放寬的幅度控制得宜，確實可以將深度優先搜尋演算法發揮極致。	IDA*演算法擁有成本花費的知識，並結合上述搜尋的演算法， 若每次放寬的幅度控制得宜，確實可以將深度優先搜尋演算法發揮極致。

p.43 的 3.3.1	以圖 3.1 和圖 3.2 爲起始盤面及圖 3.3 爲目標盤面爲例：圖 3.1，排列爲10、8、12、3、7、6、2、1、14、4、11、15、13、9、5。該對局盤的 f_1 爲51（也就是9、7、10、2、5、4、1、0、5、0、2、3、2和1的加總），f_2 爲1，故 f 爲52且是一個偶性盤面。另外以圖 3.2 爲例，則爲奇性盤面。	以圖 3.1 和圖 3.2 爲起始盤面及圖 3.3 爲目標盤面爲例：圖 3.1，排列爲10、8、12、3、7、6、2、1、14、4、11、15、13、9、5。該對局盤的 f_1 爲50（也就是9、7、9、2、5、4、1、0、5、0、2、3、2和1的加總），f_2 爲1，故 f 爲51且是一個奇性盤面。另外以圖 3.2 爲例，則爲偶性盤面。
p.44 的 圖 3.1 之圖標題	偶性盤面	奇性盤面
p.44 的圖 3.2 之圖標題	奇性盤面	偶性盤面
p.44 的圖 3.2	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>9</td><td>10</td><td>11</td><td>12</td></tr><tr><td>13</td><td>15</td><td>14</td><td></td></tr></table>	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>9</td><td>10</td><td>11</td><td>12</td></tr><tr><td>15</td><td>13</td><td>14</td><td></td></tr></table>
p.46 的 3.4 節	過去曾在1985年利用1-MIPS（型號爲 DEC 2060）的機器在每分鐘30中央處理器（central processing unit，簡稱CPU）解數個隨機出題的15-puzzle問題，其解答使用IDA*並使用曼哈頓距離爲啓發式函數。	過去曾在1985年利用1-MIPS（型號爲 DEC 2060）的機器使用30分鐘的中央處理器（central processing unit，簡稱CPU）時間解數個隨機出題的15-puzzle問題，其解答使用IDA*並使用曼哈頓距離爲啓發式函數。
p.52 的 3.8 節		本章先介紹 15-puzzle 的數學性質，將盤面分成奇性和偶性，不同奇、偶性的盤面無法透過移動空格達到，Sam Lloyd 在 1895 年的謎題，初始盤面是圖 3.2（奇性），目標盤面是圖 3.3 （偶性），因此無解。
p.52 的 3.8 節	本章介紹如何透過儲存事先計算出的部分解，方便於解答過程中化簡原本的問題。	隨後再介紹如何透過儲存事先計算出的部分解，方便於解答過程中化簡原本的問題。
p.56 的 4.1 節	雙人對局是一個雙方玩家藉由一來一往的方式，彼此互動並競爭的對局，這種雙方輪流的互動方式，通常不知不覺中會賦予先手玩家一定的優勢，因此如何設計出眞正公平的對局，是對局者的一大挑戰。	雙人對局是一個雙方玩家藉由一來一往的方式，彼此互動並競爭的對局，這種雙方輪流的互動方式，通常不知不覺中會賦予先手玩家一定的優勢，因此如何設計出眞正公平的對局是對局者的一大挑戰。
p.57 的末段	破解對局之前，必須對對局規則、對局方式、對局過程等有一定的認知，才能夠選擇最好的方法著手破解。	破解對局之前，必須對對局規則、對局方式、對局過程等有一定的認知，才能夠選擇最好的方法破解對局。
p.61 的 4.3 節	1900 年預測 2000 年電腦對局程式的發展。	1990 年預測 2000 年電腦對局程式的發展。
p.69 的定理 4.1（1）	對局屬於對稱型對局（symmetric game），也就是說先後手方的勝利條件除了顏色和方向不同外；	對局屬於對稱型對局（symmetric game），也就是說先後手方的勝利條件除了顏色和方向不同，其於都相同；
p.69 的定理 4.1（3）	對先手方而言，隨機落下一手己方的棋子不會帶給先手方任何損失，而且先手方不可能找不到一手不會使其輸棋的隨機著手；	對先手方而言，除了盤面已下滿，才可隨機落下不會造成損失的一手；

p.69 的證明（2）	六貫棋的優勝條件是：只要有一方將盤面上屬於己方顏色的邊連成一線即爲獲勝，但先手方獲勝後必須給後手方最後一擊的機會，若後手方也連成一串，則後手方同時間也獲勝，盤面上同色的邊彼此平行，在一方取得獲勝資格的同時也意味著盤面可以沿著同色棋子連成的線分割成兩半，另一方必須要跨越此線才能夠取得勝利資格，但這是不可能的，因此不會出現雙方一起獲得勝利的情況。	六貫棋的優勝條件是：只要有一方將盤面上屬於己方顏色的邊連成一線即爲獲勝，但先手方獲勝後必須給後手方最後一擊的機會，若後手方也連成一串，則後手方同時間也獲勝，盤面上同色的邊彼此平行，在一方取得獲勝資格的同時也意味著盤面可以沿著同色棋子連成的線分割成兩半，另一方不可能跨越此線以取得勝利資格，因此不會出現雙方一起獲得勝利的情況。
p.69 的證明（3）	假設在盤面上 x 位置處落下先手方一子，若此位置剛好是位在先手方必勝的路徑上，則先手方可以改落其他位置；若此位置不在先手方必勝的路徑上，則先手方可以忽略此步。	假設要在盤面上某位置處落下先手方一子，如果先手方本來可以贏棋，不會因爲此手而不贏棋。對手也不會因此而由不贏棋變成贏棋。
p.71 的圖 4.4 之圖標題	六貫棋性質證明：連接黑方棋子	六貫棋性質證明：連接各行中的黑方棋子
p.71 的 4.5.1 節	依定理 4.1 我們可知，六貫棋對局一定會結束、不可能出現兩個玩家同時獲勝的情況，以及不可能出現和棋的三個性質。	依定理 4.1 我們可知，六貫棋對局一定會結束、不可能出現兩個玩家同時獲勝的情況，以及不可能出現和棋的三個性質[8]。
p.71 附註		[8] 另注意本節很多棋串連接特性和六連接有關，可能不適用於其他棋類例如圍棋之四連接。
p.72 的 4.5.2 節	下一回合起，先手方依舊將盤面上隨機產生的棋子位置 m' 去除，利用 $rev(f(B_0{+}rev(m_2){+}m_3{+}rev(m_4)))$ 佯裝成後手查詢必勝著手。	下一回合起，先手方依舊將盤面上隨機產生的棋子位置 m' 去除，利用 $rev(f(B_0{+}rev(m_2){+}rev(m_3){+}rev(m_4)))$ 佯裝成後手查詢必勝著手。
p.73 的 4.5.2 節		3. 爲對稱型對局，且剛好只有一位玩家。先手不會第一手便輸。
p.74 的 4.6.1 節	以2015年電腦硬體的發展程度爲例，狀態空間如果落在 $10^{13} \sim 10^{14}$ 之間可以利用窮舉法破解。	以2015年電腦硬體的發展程度爲例，狀態空間如果落在 $10^{13} \sim 10^{14}$ 之間的雙人對局可以利用窮舉法破解。
p.74 的 4.6.1 節	以目前電腦硬體的發展，若對局樹的深度落在30左右，大多可以利用搜尋程式破解，如使用較豐富的棋局知識可以大幅裁減對弈時產生的對局樹，就連一些深度更深的對局也可以被解出。	以目前電腦硬體的發展，若對局樹的深度落在30左右，大多可以利用搜尋程式破解。如使用較豐富的棋局知識可以大幅裁減對弈時產生的對局樹，就連一些深度更深的對局也可以被解出。
p.75 的 4.6.1 節	Category 4：狀態空間複雜度和對局樹複雜度都比其他對局高，目前未有較佳的解法，在圍棋上採用蒙地卡羅模擬法似乎帶來不少解決此類問題的希望。	Category 4：狀態空間複雜度和對局樹複雜度都比其他對局高，目前未有較佳的解法。在圍棋上採用蒙地卡羅模擬法似乎帶來不少解決此類問題的希望。
p.75 的 4.6.1 節	此表格對於對局破解程度的想法，主要是根據對局複雜度，對局複雜度可以根據對局狀態空間和對局樹的大小分爲四個類型。	此表格對於對局破解程度的想法，主要是根據對局複雜度。對局複雜度可以根據對局狀態空間和對局樹的大小分爲四個類型。
p.75 的 4.6.2 節	一般而言，公平型對局較容易被玩家接受，但是因爲先手優勢的因素，很多對局變得不公平。	一般而言，公平型對局較容易被玩家接受。但是因爲先手優勢的因素，很多對局變得不公平。
p.76 的首段	先手優勢的收回方式，根據對局性質與特色會有不同的處理方式。	如何以規則中和先手優勢，根據對局性質與特色通常有不同的處理方式。

p.89 的圖 5.7 之圖標題	位在中央的騎士	騎士的影響
p.91 的圖 5.14 之圖標題	栓鏈的範例	栓鏈
p.103 的 6.1.1 節	在搜尋樹中，一個節點的值就是其相應的位置的值。	
p.106 的 6.2 節	（2）目前的搜尋深度抵達預設的搜尋深度；	（2）剩下的待搜尋深度[1]*depth*已經降爲0；
p.106 附註		爲簡化本書敘述，若呼叫函數時之 *depth* 參數並非討論之重點，則會省略不寫。
p.107 的 Algorithm 15	**Algorithm 15:** F′(position p)	**Algorithm 15:** F′(position p, integer *depth*)
p.107 的 Algorithm 15	3 **or** depth reaches the cut-off threshold // iterative deepening	3 **or** depth = 0 // remaining depth to search
p.107 的 Algorithm 15	11 t := G′(p_i);	11 t := G′(p_i, *depth*-1);
p.108 的 Algorithm 16	**Algorithm 16:** G′(position p)	**Algorithm 16:** G′(position p, integer *depth*)
p.108 的 Algorithm 16	3 **or** depth reaches the cut-off threshold // iterative deepening	3 **or** depth = 0 // remaining depth to search
p.108 的 Algorithm 16	11 t := F′(p_i);	11 t := F′(p_i, *depth*-1);
p.111 的 Algorithm 17	**Algorithm 17:** F(position p)	**Algorithm 17:** F(position p, integer *depth*)
p.111 的 Algorithm 17	3 **or** depth reaches the cut-off threshold // iterative deepening	3 **or** depth = 0 // remaining depth to search
p.111 的 Algorithm 17	11 t := -F(p_i); // recursive call, the returned value is negated	11 t := -F(p_i, *depth*-1); // recursive call, the returned value is negated
p.111 的 6.4.1 節	比方說目前搜尋的結果是 x 分，當確定某一個分枝沒有辦法給出比 x 還要好的分數時，就無須對這個分枝進行額外的搜尋，因爲這個即將被剪枝的分枝無法回傳優於目前最佳值 x。	比方說目前搜尋的結果是 x 分，當確定某一個分枝沒有辦法給出比 x 還要好的分數時，就無須對這個分枝進行額外的搜尋，因爲這個即將被剪枝的分枝無法回傳優於目前最佳值 x 的結果或數值。
p.112 的 6.4.1 節	和對局樹有完全相同的值，但是使用不同的搜尋順序會產生截然不同的剪枝行爲。	對局樹有完全相同的值，但是使用不同的搜尋順序會產生截然不同的剪枝行爲。
p.112 的 6.4.2 節	在圖 6.4 中，根節點在搜尋完子節點 1 後得到目前的最佳值15，由於在搜尋的過程中，最大化節點的目前最佳值只會不斷上升而不會下降，因此可以將目前的最佳值15當作一個邊界值（bound），這個邊界值保證了根節點的最佳值的下界，也就是眞正的最佳值至少會和邊界值相同。	在圖 6.4 中，根節點在搜尋完子節點 1 後得到目前的最佳值15，由於在搜尋的過程中，最大化節點的目前最佳值只會上升而不會下降，因此可以將目前的最佳值15當作一個邊界值（bound），這個邊界值保證了根節點的最佳值的下界，也就是眞正的最佳值至少會和邊界值相同。
p.114 的 6.4.3 節	節點1首先搜尋了節點1.1得到其回傳值10，由於節點 1是最小化節點因此可以知道，無論節點 1.2 的回傳值是多少，節點1的最佳值都只會比 10 小，也就是節點1目前的上界值爲 10，一般把最小化節點的上界值稱爲 *beta*。	節點1首先搜尋了節點1.1得到其回傳值10，由於節點 1 是最小化節點，因此可以知道，無論節點 1.2 的回傳值是多少，節點 1 的最佳值都只會比10小，也就是節點1目前的上界值爲10。一般把最小化節點的上界值稱爲 *beta*。

p.115 的 6.4.4 節	假設最小化節點 v 的父節點 u 在搜尋 v 之前，先搜尋 u 的其他子節點（v 的兄長）並提供一個下界值 V_L 給身爲最大化節點的 u，也就是 u 眞正的值$V_u \geq V_L$。而節點 v 的第一個子節點 w 提供了一個上界值 V_U 給身爲最小化節點的 v，也就是 v 眞正的值 $V_U \geq V_v$。	假設節點 u 在搜尋 v 之前，先搜尋 u 的其他子節點（v 的兄長）並提供一個下界值 V_L 給身爲最大化節點的 u，也就是 u 眞正的值$V_u \geq V_L$。而節點 v 的第一個子節點 w 提供了一個上界值 V_U 給身爲最小化節點的 v，也就是 v 眞正的值 $V_U \geq V_v$。
p.116 的 6.4.4 節	在圖 6.6 中，根節點首先搜尋節點1得到了下界值 V_L =15。接著根節點依照節點2、2.1、2.1.1和2.1.1.1的順序依次搜尋，在節點2.1.1得到了一個上界值 V_U =7。	在圖 6.6 中，根節點首先搜尋節點1得到了下界值 V_L =15。接著根節點依照節點2、2.1、2.1.1和2.1.1.1的順序依次搜尋，在節點2.1.1.1得到了一個上界值 V_U=7。
p.117 的 6.4.5 節	加上 Alpha-Beta 切捨後，原本的最小最大搜尋演算法被改變了，變成具有 Alpha-Beta 切捨功能的演算法 F2′（Algorithm 18）和演算法 G2′（Algorithm 19）。	加上 Alpha-Beta 切捨後，原本的最小最大搜尋演算法被改變了，變成具有 Alpha-Beta 切捨功能的演算法 F1′（Algorithm 18）和演算法 G1′（Algorithm 19）。
p.117 的 6.4.5 節	接著以圖 6.4 爲例，演示演算法 F2′ 和 G2′ 的執行過程。由於根節點是最大化節點，整個搜尋過程由呼叫函式 $F2'$(root, $-\infty$, ∞) 開始，執行之初，m 被初始化成 $-\infty$。接著程式呼叫 $G2'$(node 1, $-\infty$, ∞) 以計算節點1的值。因爲節點1是終端節點，於是節點1立刻回傳了其值15給 F2′。現在 t 的值變成了15。因爲 t 比目前的最大值 $-\infty$ 大，於是更新目前的最大值 m 成爲 t 的值15，由於 m 的值15沒有比目前的上界值 ∞ 大，因此不進行剪枝，接著 $F2'$(root, $-\infty$, ∞) 呼叫 $G2'$(node 2, 15, ∞) 以計算節點2的值。在 G2′ 中，節點2呼叫 $F2'$(node 2.1, 15, ∞) 以計算節點2.1的值，由於節點2.1是終端節點，於是回傳其值10給 G2′。在 G2′ 中，由於10比目前節點2中的最小值 ∞ 小，因此目前的最小值被更新爲10。此時，G2′ 中的目前最小值10比下界值 *alpha* 的值15還要小，因此產生Alpha剪枝。$G2'$(node 2, 15, ∞) 捨棄節點2.2及2.3不進行搜尋，	接著以圖 6.4 爲例，演示演算法 F1′ 和 G1′ 的執行過程。由於根節點是最大化節點，整個搜尋過程由呼叫函式 $F1'$(root, $-\infty$, ∞) 開始，執行之初，m 被初始化成 $-\infty$。接著程式呼叫 $G1'$(node 1, $-\infty$, ∞) 以計算節點1的值。因爲節點1是終端節點，於是節點1立刻回傳了其值15給 F1′。現在 t 的值變成了15。因爲 t 比目前的最大值 $-\infty$ 大，於是更新目前的最大值 m 成爲 t 的值15，由於 m 的值15沒有比目前的上界值 ∞ 大，因此不進行剪枝，接著 $F1'$(root, $-\infty$, ∞) 呼叫 $G1'$(node 2, 15, ∞) 以計算節點2的值。在 G1′ 中，節點2呼叫 $F1'$(node 2.1, 15, ∞) 以計算節點2.1的值，由於節點2.1是終端節點，於是回傳其值10給 G1′。在 G1′ 中，由於10比目前節點2中的最小值 ∞ 小，因此目前的最小值被更新爲10。此時，G1′ 中的目前最小值10比下界值 *alpha* 的值15還要小，因此產生Alpha剪枝。$G1'$(node 2, 15, ∞) 捨棄節點2.2及2.3不進行搜尋，
p.118 的 Algorithm 18 之標題	F2′(position p, value *alpha*, value *beta*)	F1′(position p, value *alpha*, value *beta*)
p.118 的 Algorithm 18	11 return m; // Beta cut-off	11 return *beta*; // Beta cut-off
p.118 的 Algorithm 19 之標題	G2′(position p, value *alpha*, value *beta*)	G1′(position p, value *alpha*, value *beta*)
p.118 的 Algorithm 19	7 $t := F2'(p_i, alpha, m)$;	7 $t := F1'(p_i, alpha, m)$;
p.118 的 Algorithm 19	11 return m; // Alpha cut-off	11 return *alpha*; // Alpha cut-off
p.119 的 6.4.5 節	直接回傳目前的最小值10給 $F2'$(root, $-\infty$, ∞)。	直接回傳目前的*alpha*值15給 $F1'$(root, $-\infty, \infty$)。

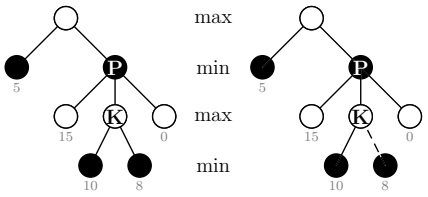
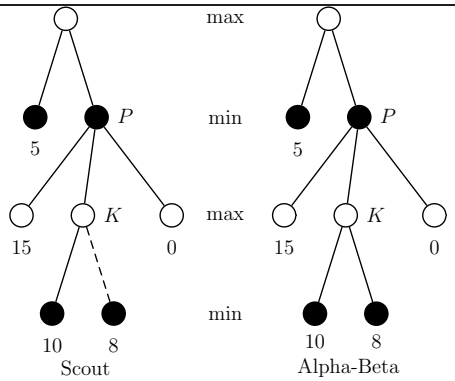
p.119 的 6.4.5 節	如同將最小最大化（Minimax）版本的演算法F′及G′轉成正反最大（Negamax）版的演算法F，同樣地也可以將最小最大化版本的演算法F2′ 及G2′ 轉換爲正反最大版的演算法，最終轉換而成的，便是正反最大版的演算法F2（Algorithm 20）。	如同將最小最大化（Minimax）版本的演算法 F′ 及 G′轉成正反最大（Negamax）版的演算法F，同樣地也可以將最小最大化版本的演算法 F1′及G1′轉換爲正反最大化版的 Alpha-Beta 切捨演算法，最終轉換而成的，便是正反最大化版的Alpha-Beta切捨演算法 F1（詳細演算法在此省略），演算法 F2（Algorithm 20）則只在第 15 行修改回傳值爲m而不是 F1 中的$beta$， 此修改版將在 6.6 節中介紹。
p.119 的 Algorithm 20	**Algorithm 20:** F2(position p, value $alpha$, value $beta$)	**Algorithm 20:** F2(position p, value $alpha$, value $beta$, integer $depth$)
p.119 的 Algorithm 20	3 **or** depth reaches the cut-off threshold // iterative deepening	3 **or** depth $= 0$ // remaining depth to search
p.119 的 Algorithm 20	5 **or** some other constraints are met // add knowledgei here	5 **or** some other constraints are met // add knowledge here
p.119 的 Algorithm 20	11 t := -F2(p_i, -$beta$, -m);	11 t := -F2(p_i, -$beta$, -m, $depth$-1);
p.123 的 6.5.2 節	（1）根節點的第一個分枝及其下的節點的第一個分枝都必須被搜尋直到達到終端盤面爲止；	（1）根節點的第一個分枝及其下節點的第一個分枝都必須被搜尋直到達到終端盤面爲止；
p.126 的末段	（2）因爲 $(\ell-1)-j$ 是奇數且 $a_j \neq 1$，所以 $a_{\ell-1} = 1$。	（2）由於 $(\ell-1)-j$ 是奇數且 $a_j \neq 1$，所以 $a_{\ell-1} = 1$。
p.129 的 6.5.3 首段	這個小節將利用 6.5.2 節所介紹的關鍵節點的性質，證明定理 6.1，也就是在一棵完美排序樹中，所有的關鍵節點皆只會被 Alpha-Beta 切捨演算法拜訪，且會被 Alpha-Beta 切捨演算法拜訪的節點亦會是關鍵節點。	這個小節將利用 6.5.2 節所介紹的關鍵節點的性質，證明定理 6.1，也就是在一棵完美排序樹中，所有的關鍵節點皆會被 Alpha-Beta 切捨演算法拜訪，且會被 Alpha-Beta 切捨演算法拜訪的節點亦會是關鍵節點。
p.129 的 6.5.3 節	首先，論述三種類的關鍵節點分別在何時被拜訪：第1類節點在呼叫$F2(p, -\infty, \infty)$時被拜訪；第2類節點在呼叫$F2(p, -\infty, beta)$ 時被拜訪；第3類節點在呼叫$F2(p, alpha, \infty)$ 時被拜訪。	首先，論述三種類的關鍵節點分別在何時被拜訪：第1類節點在呼叫$F1(p, -\infty, \infty, depth)$時被拜訪；第2類節點在呼叫$F1(p, -\infty, beta, depth)$ 時被拜訪；第3類節點在呼叫$F1(p, alpha, \infty, depth)$ 時被拜訪。

p.129 的 6.5.3 節	當 $F\underline{2}(p,-\infty,\infty)$ 被呼叫時，它會繼續呼叫 $F\underline{2}(p_1,-\infty,\infty)$，拜訪 p 的第一個子節點 p_1。第一個被呼叫的 $F\underline{2}(p,-\infty,\infty)$ 中的 p 是根節點，而根節點是第1類節點，又第1類節點的第一子節點也是第1類節點，因此在一連串的 $F\underline{2}(p,-\infty,\infty)$呼叫中，會把所有的第1類節點都拜訪過一次。因爲目前拜訪的樹是完美排序樹，故可歸納出：$F(p)= -F(p_1)\neq \pm\infty$。在 p 的第一個子節點將值回傳之後，由於 F$\underline{2}$ 總是更新 $alpha$ 值而使用 $beta$ 值進行剪枝，在 $F\underline{2}(p,-\infty,\infty)$ 中，$beta$ 值是 ∞ 的情況下是不會發生剪枝的。因此，$F\underline{2}(p,-\infty,\infty)$ 會用 p_1 回傳的值更新 $alpha$ 值並呼叫 $F\underline{2}(p_i,-\infty,F(p_1))$ 計算下一個分枝 p_i，這邊注意到由於之後的每一個p_i都只會嘗試更新$F\underline{2}(p,-\infty,\infty)$中的$alpha$值，因此都不會發生剪枝，也就是所有的$p_i$都會分別被$F\underline{2}(p,-\infty,\infty)$中呼叫的$F\underline{2}(p_i,-\infty,F(p_1))$所拜訪，按照定義，這些 p_i 都是第2.1類的節點。	當 $F\underline{1}(p,-\infty,\infty,\underline{depth})$ 被呼叫時，它會繼續呼叫 $F\underline{1}(p_1,-\infty,\infty,\underline{depth})$，拜訪 p 的第一個子節點 p_1。第一個被呼叫的 $F\underline{1}(p,-\infty,\infty,\underline{depth})$ 中的 p 是根節點，而根節點是第1類節點，又第1類節點的第一子節點也是第1類節點，因此在一連串的 $F\underline{1}(p,-\infty,\infty,\underline{depth})$呼叫中，會把所有的第1類節點都拜訪過一次。因爲目前拜訪的樹是完美排序樹，故可歸納出：$F(p)= -F(p_1)\neq \pm\infty$。在 p 的第一個子節點將值回傳之後，由於 F$\underline{1}$ 總是更新 $alpha$ 值而使用 $beta$ 值進行剪枝，在 $F\underline{1}(p,-\infty,\infty,\underline{depth})$ 中，$beta$ 值是 ∞ 的情況下是不會發生剪枝的。因此，$F\underline{1}(p,-\infty,\infty,\underline{depth})$ 會用 p_1 回傳的值更新 $alpha$ 值並呼叫 $F\underline{1}(p_i,-\infty,F(p_1),\underline{depth})$ 計算下一個分枝 p_i，這邊注意到由於之後的每一個p_i都只會嘗試更新$F\underline{1}(p,-\infty,\infty,\underline{depth})$ 中的$alpha$值，因此都不會發生剪枝，也就是所有的p_i都會分別被$F\underline{1}(p,-\infty,\infty,\underline{depth})$中呼叫的$F\underline{1}(p_i,-\infty,F(p_1),\underline{depth})$所拜訪，按照定義，這些 p_i 都是第$\underline{2.1}$類的節點。
p.130 的 6.5.3 節	考慮一個第2類節點p_i被$F\underline{2}(p_i,-\infty,beta)$拜訪的當下。其中 p_i 的父節點是 p 而 p_i 是 p 的第 i 個子節點且 $i\neq 1$。由於這是一棵完美排序樹，因此 $F\underline{2}(p_i,-\infty,beta)$ 中的 $beta = F(p_1)$，對於所有的 i 而言，有 $-F(p_1)\geq -F(p_i)$ 也就是 $beta = F(p_1)\leq F(p_i)$。又因爲完美排序樹的關係，$F(p_i)= -F(p_{i.1})$，因此當 p_i 的第一個子節點回傳$-F(p_{i.1})$給p_i時，就會因爲$beta = F(p)\leq F(p_i)= -F(p_{i.1})$而產生剪枝。因此所有的第2類節點被 $F\underline{2}(p_i,-\infty,beta)$ 呼叫時，只會呼叫 $F\underline{2}(p_{i.1},-beta,\infty)$ 拜訪它的第一個子節點。而第2類節點的第一個子節點正好是第3類節點。	考慮一個第2類節點p_i被$F\underline{1}(p_i,-\infty,beta,depth)$ 拜訪的當下。其中 p_i 的父節點是 p 而 p_i 是 p 的第 i 個子節點且 $i\neq 1$。由於這是一棵完美排序樹，因此 $F\underline{1}(p_i,-\infty,beta,depth)$ 中的 $beta = F(p_1)$，對於所有的 i 而言，有 $-F(p_1)\geq -F(p_i)$ 也就是 $beta = F(p_1)\leq F(p_i)$。又因爲完美排序樹的關係，$F(p_i)= -F(p_{i.1})$，因此當 p_i 的第一個子節點回傳 $-F(p_{i.1})$ 給 p_i 時，就會因爲 $beta = F(p) \leq F(p_i) = -F(p_{i.1})$ 而產生剪枝。因此所有的第 2 類節點被 $F\underline{1}(p_i,-\infty,beta,depth)$ 呼叫時，只會呼叫 $F\underline{1}(p_{i.1},-beta,\infty,depth)$ 拜訪它的第一個子節點。而第2類節點的第一個子節點正好是第3類節點。

p.130 的 6.5.3 節	接著考慮第3類子節點被 $F\underline{2}(p,alpha,\infty)$ 拜訪當下發生的所有狀況。由於目前的 $beta$ 值爲 ∞，因此不會產生任何剪枝，也就是p 會拜訪它所有的子節點，而一個第3類節點的所有子節點正好是第2類子節點。故它的子節點p_1、p_2、$\cdots\cdots$、p_b會分別被$F\underline{2}(p_1,-\infty,-alpha)$、$F\underline{2}(p_2,-\infty,-\max\{m_1,alpha\})$、$\cdots\cdots$、$F\underline{2}(p_b,\ -\infty,\ -\max\{m_{b-1},\ alpha\})$拜訪，其 m_i爲搜尋完前 i 個分枝後的最佳值。當 $i \geq 2$ 時， $m_i\ =\ F\underline{2}(p_i,\ -\infty,\ -\max\{m_{i-1},\ alpha\})$	接著考慮第3類子節點被 $F\underline{1}(p,alpha,\infty,\underline{depth})$ 拜訪當下發生的所有狀況。由於目前的 $beta$ 值爲 ∞，因此不會產生任何剪枝，也就是p會拜訪它所有的子節點，而一個第3類節點的所有子節點正好是第2類子節點。故它的子節點p_1、p_2、$\cdots\cdots$、p_b會分別被$F\underline{1}(p_1,-\infty,-alpha,\underline{depth})$、$F\underline{1}(p_2,-\infty,-\max\{m_1,alpha\},\underline{depth})$、$\cdots\cdots$、$F\underline{1}(p_b,-\infty,-\max\{m_{b-1},alpha\},\underline{depth})$拜訪，其 m_i爲搜尋完前 i 個分枝後的最佳值。當 $i \geq 2$ 時， $m_i\ =\ F\underline{1}(p_i,-\infty,-\max\{m_{i-1},alpha\}\ ,\underline{depth})$
p.130 的 6.5.3 節	綜合以上論述，F$\underline{2}$在一棵完美排序樹中，會拜訪的節點就是所有的關鍵節點，而所有的關鍵節點在F$\underline{2}$中也一定會被拜訪。因此定理 6.1 得以被證明。經由定理 6.1，可以得到推論 6.1。	綜合以上論述，F$\underline{1}$在一棵完美排序樹中，會拜訪的節點就是所有的關鍵節點，而所有的關鍵節點在F$\underline{1}$中也一定會被拜訪。因此定理 6.1 得以被證明。經由定理 6.1，可以得到推論 6.1。
p.130 的證明	1. 所有 k 爲$\underline{\text{奇數}}$，$a_k = 1$ 的路徑有 $b^{\lceil i/2 \rceil}$ 個；	1. 所有 k 爲$\underline{\text{偶數}}$，$a_k = 1$ 的路徑有 $b^{\lceil i/2 \rceil}$ 個；
p.130 的證明	2. 所有 k 爲$\underline{\text{偶數}}$，$a_k = 1$ 的路徑有 $b^{\lfloor i/2 \rfloor}$ 個；	2. 所有 k 爲$\underline{\text{奇數}}$，$a_k = 1$ 的路徑有 $b^{\lfloor i/2 \rfloor}$ 個；
p.131 的 6.5.3 節	因此在第 i 層被拜訪的節點數是奇數分枝爲1的關鍵節點加上偶數分枝爲1的關鍵節點，$\underline{\text{再減去被重複計算奇數偶數皆爲1的關鍵節點：}}$ $b^{\lceil i/2 \rceil} + b^{\lfloor i/2 \rfloor} - 1$	因此在第 i 層被拜訪的節點數是奇數分枝爲1的關鍵節點加上偶數分枝爲1的關鍵節點，$\underline{\text{再減去}}$ $b^{\lceil i/2 \rceil} + b^{\lfloor i/2 \rfloor} - 1$ $\underline{\text{個被重複計算奇數偶數皆爲1的關鍵節點。}}$
p.131 的 6.5.3 節	藉由推論 6.1 可以知道若樹的深度是 ℓ，則$\underline{\text{整棵樹被拜訪的節點數一共是：}}$ $\sum_{i=0}^{\ell} b^{\lceil i/2 \rceil} + b^{\lfloor i/2 \rfloor} - 1$	藉由推論 6.1 可以知道若樹的深度是 ℓ，則 $\sum_{i=0}^{\ell} b^{\lceil i/2 \rceil} + b^{\lfloor i/2 \rfloor} - 1$ $\underline{\text{爲整棵樹被拜訪的節點數。}}$
p.135 的 6.6 節	在本章中，首先介紹了暴力式正法最大（brute-force Negamax）的F演算法，這一個版本的演算法依照負最大的公式，總是會得到正確值。其後介紹的是$\underline{\text{硬式失敗}}$版本Alpha-Beta$\underline{\text{切捨}}$演算法$\underline{\text{（fail hard Alpha-Beta pruning algorithm）}}$，在本節接著要介紹的是F$\underline{2}$的$\underline{\text{一}}$個變形：軟式失敗版本Alpha-Beta切捨演算法（fail soft Alpha-Beta pruning algorithm）F3。$\underline{\text{在正式介紹F3之前，首先回顧硬式失敗版的演算法F2。}}$	在本章中，首先介紹了暴力式正法最大（brute-force Negamax）的F演算法，這一個版本的演算法依照負最大的公式，總是會得到正確值。其後介紹的是$\underline{\text{原}}$版本Alpha-Beta切捨演算法F$\underline{1}$，在本節接著要介紹的是F$\underline{1}$的$\underline{\text{兩}}$個變形：$\underline{\text{硬式失敗版}}$F2$\underline{\text{（fail hard Alpha-Beta pruning algorithm）和軟式失敗版}}$本Alpha-Beta切捨演算法（fail soft Alpha-Beta pruning algorithm）F3。

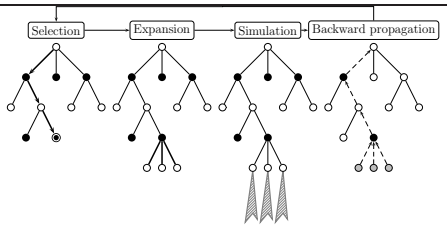
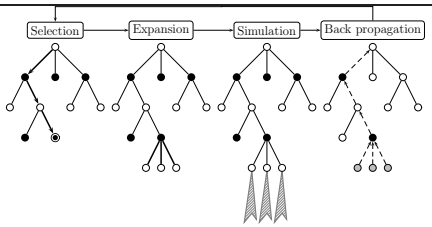
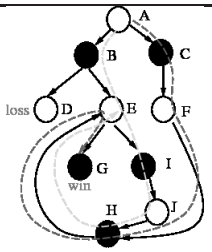
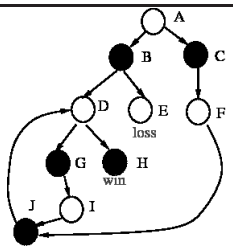
p.135 的 6.6.1 節	通常在計算根節點 r 時，會呼叫 $F2(r, -\infty,\infty)$，也就是在搜尋區間 $[-\infty,\infty]$ 進行搜尋。如果改用非正負無限大的搜尋區間 $[alpha, beta]$ 進行搜尋，也就是呼叫 $F2(r, alpha, beta)$ 時，代表什麼意思呢？原始的 $F2(r, -\infty, \infty)$ 沒有對 r 的值作任何假設，會在正負無限大的區間中進行搜尋，而使用 $F2(r, alpha, beta)$ 進行搜尋則表示預期 r 的值會落在區間 $[alpha, beta]$ 之中。	通常在計算根節點 r 時，會呼叫 $F2(r, -\infty,\infty, depth)$，也就是在搜尋區間 $[-\infty,\infty]$ 進行搜尋。如果改用非正負無限大的搜尋區間 $[alpha, beta]$ 進行搜尋，也就是呼叫 $F2(r,alpha,beta, depth)$ 時，代表什麼 意思呢？原始的 $F2(r, -\infty, \infty ,depth)$ 沒有對 r 的值作任何假設，會在正負無限大的區間中進行搜尋，而使用 $F2(r, alpha, beta, depth)$ 進行搜尋則表示預期 r 的值會落 在區間 $[alpha, beta]$ 之中。
p.136 的 6.6.1 節	當給定 $alpha$ 和 $beta$（且滿足 $alpha \leq beta$），而且目前搜尋的位置 p 爲非終端節點時，$F2(p, alpha, beta)$ 與 $F(p)$ 的大小關係一共有以下三種可能： 1. 當 $F(p) \leq alpha$ 時，$F2(p, alpha, beta) = alpha$ 2. 當 $alpha < F(p) < beta$ 時，$F2(p, alpha, beta) = F(p)$ 3. 當 $F(p) \geq beta$ 時，$F2(p, alpha, beta) \geq beta$ 且 $F(p) \geq F2(p, alpha, beta)$ 上述的分類表示，$F2(p, alpha, beta)$ 會依照正反最大的公式在給定的區間 $[alpha, beta]$ 中尋找最好的值。倘若$F(p)$ 的值比 $alpha$ 來得小，則 $F2(p, alpha, beta)$ 的回傳值 $alpha$ 是從某個值比 $alpha$ 小的終端節點回傳而來的，同理，倘若$F(p)$ 的值比 $beta$ 來得大，則 $F2(p, alpha, beta)$的回傳值會是從某個值比 $beta$ 大的終端節點回傳而來的。綜上所述，當 $alpha = -\infty$ 且 $beta = \infty$ 時，由於 $-\infty < F(p) < \infty$，因此，$F2(p, -\infty, +\infty)$ 亦總是回傳正確的 $F(p)$ 值。	當給定 $alpha$ 和 $beta$（且滿足 $alpha \leq beta$），而且目前搜尋的位置 p 爲非終端節點時，$F2(p, alpha, beta, depth)$ 與 $F(p)$ 的大小關係一共有以下三種可能： 1. 當 $F(p) \leq alpha$ 時，$F2(p, alpha, beta, depth) = alpha$ 2. 當 $alpha < F(p) < beta$ 時，$F2(p, alpha, beta, depth) = F(p)$ 3. 當 $F(p) \geq beta$ 時，$F2(p, alpha, beta, depth) \geq beta$ 且 $F(p) \geq F2(p, alpha, beta, depth)$ 上述的分類表示，$F2(p, alpha, beta, depth)$ 會依照正反最大的公式在給定的區間 $[alpha, beta]$ 中尋找最好的值。倘若$F(p)$ 的值比 $alpha$ 來得小，則 $F2(p, alpha, beta, depth)$ 的回傳值 $alpha$ 是從某個值比 $alpha$ 小的終端節點回傳而來的，同理，倘若$F(p)$ 的值比 $beta$ 來得大，則 $F2(p, alpha, beta, depth)$的回傳值會大於$beta$且是從某個值比 $beta$ 大的終端節點回傳而來的。綜上所述，當 $alpha = -\infty$ 且 $beta = \infty$ 時，由於 $-\infty < F(p) < \infty$，因此，$F2(p, -\infty, +\infty, depth)$ 亦總是回傳正確的 $F(p)$ 值。
p.136 的 6.6.1 節	而F2演算法總是回傳一個位於區間 $[alpha, beta]$ 中的值。	而F2演算法 總是回傳一個位於區間 $[alpha, beta]$ 中的值，但F2回傳值必$\geq alpha$但也可能$> beta$。
p.136 的 6.6.1 節	以圖 6.14 爲例，一開始的搜尋區間是[4000, 5000]，節點 A 呼叫$F2(W, -5000,-4000)$ 計算節點 W 的值，而後節點 W 回傳了值 -200給節點 A，其後節點 A 呼叫 $F2(Q, -5000, -4000)$ 計算節點 Q 的值…….	以圖 6.14 爲例，一開始的搜尋區間是[4000, 5000]，節點 A 呼叫$F2(W, -5000,-4000,depth)$ 計算節點 W 的值，而後節點 W 回傳了值 -200給節點 A，其後節點 A 呼叫 $F2(Q, -5000, -4000, depth)$ 計算節點 Q 的值…….

p.137-138 的 6.6.2 節	在 $alpha < beta$ 以及目前搜尋的位置 p 爲非終端位置的條件下，$F3(p, alpha, beta)$ 的回傳值與 $F(p)$ 之間的關係，一共有以下三種可能： 1. 當 $F(p) \leq alpha$ 時，$F3(p, alpha, beta) \leq alpha$且$F(p) \leq F3(p, alpha, beta)$ 2. 當 $alpha < \mathrm{F}(p) < beta$ 時，$F3(p, alpha, beta) = F(p)$ 3. 當 $F(p) \geq beta$ 時，$F3(p, alpha, beta) \geq beta$ 且 $F(p) \geq F3(p, alpha, beta)$	在 $alpha < beta$ 以及目前搜尋的位置 p 爲非終端位置的條件下，$F3(p, alpha, beta, depth)$ 的回傳值與 $F(p)$ 之間的關係，一共有以下三種可能： 1. 當 $F(p) \leq alpha$ 時，$F3(p, alpha, beta, depth) \leq alpha$且$F(p) \leq F3(p, alpha, beta, depth)$ 2. 當 $alpha < \mathrm{F}(p) < beta$ 時，$F3(p, alpha, beta, \mathrm{depth}) = F(p)$ 3. 當 $F(p) \geq beta$ 時，$F3(p, alpha, beta, \infty) \geq beta$ 且 $F(p) \geq F3(p, alpha, beta, \infty)$
p.138 的 Algorithm 21	F3(position p, value $alpha$, value $beta$)	F3(position p, value $alpha$, value $beta$, value $depth$)
p.138 的 Algorithm 21	3 **or** depth reaches the cut-off threshold //iterative deepening	3 **or** $depth = 0$ // remaining depth to search
p.138 的 Algorithm 21	11 $t :=$ $-F3(p_i, -beta, -\max\{m, alpha\})$;	11 $t := -F3(p_i, -beta, -\max\{m, alpha\}, depth - 1)$;
p.138 的 6.6.2 節	比較F2和F3的回傳值可以發現：當上界失敗發生時，F3的回傳值和F2的回傳值都會比 $beta$ 高，但不會高過眞正的 $F(p)$，當下界失敗發生時，F3的回傳值則會比F2的回傳值來得低，但不會低過眞正的 $F(p)$。而當上界失敗和下界失敗皆未發生時，如同 $F2(p, -\infty, \infty)$ 總是回傳正確的 $F(p)$ 值，$F3(p, -\infty, +\infty)$ 亦總是回傳正確的 $F(p)$ 值。	比較F2和F3的回傳值可以發現：當上界失敗發生時，F3的回傳值和F2的回傳值都會比 $beta$ 高，但不會高過眞正的 $F(p)$，當下界失敗發生時，F3的回傳值則會比F2的回傳值來得低，但不會低過眞正的 $F(p)$。而當上界失敗和下界失敗皆未發生時，如同 $F2(p, -\infty, \infty, \infty)$ 總是回傳正確的 $F(p)$ 值，$F3(p, -\infty, +\infty, \infty)$ 亦總是回傳正確的 $F(p)$ 值。
p.138 的 6.6.2 節	以圖 6.15 爲例，一開始的搜尋區間是[4000, 5000]，節點 A 呼叫 $F3(W, -5000, -4000)$ 計算節點 W 的值，而後節點 W 回傳了值 -200 給節點 A，其後，節點 A 呼叫 $F3(Q, -5000, -4000)$ 計算節點 Q 的值，節點 Q 回傳 $-v$ 給節點 A，最後，節點 A 回傳 200和 v 中的最大者。在這之中可以發現，假設節點 W 的值爲 u，只要 u 比目前的下界值 $alpha$ 還要小，則節點 A 的回傳值就至少比節點 W 的回傳值 u 大。	以圖 6.15 爲例，一開始的搜尋區間是[4000, 5000]，節點 A 呼叫 $F3(W, -5000, -4000, depth)$ 計算節點 W 的值，而後 W 回傳了值-200 給 A，其後，節點 A 呼叫 $F3(Q, -5000, -4000, depth)$ 計算節點 Q 的值，節點 Q 回傳$-v$ 給A，最後，節點 A 回傳 200和 v 中的最大者。在這之中可以發現，假設節點 W 的值爲 u，只要 u 比目前的下界值 $alpha$ 還要小，則節點 A 的回傳值就至少比節點 W 的回傳值 u 大。
p.146 的 7.2.2 節	在節點 1.1.1將結果回傳給節點 1.1時，由於節點 1.1是最大化節點，在測試 $> v$ 時，只要有任一子節點回傳 $true$，其結論亦會是 $true$。因此可以不用詢問節點 1.1.2和1.1.3的結果。	在節點 1.1.1將結果回傳給節點 1.1時，由於節點 1.1是最大化節點，在測試 $> v$ 時，只要有任一子節點回傳 $true$，其結論亦會是 $true$，因此可以不用詢問節點 1.1.2和1.1.3的結果。

p.151 的圖7.3		
p.152 的圖 7.4 之圖標題	斥候演算法搜尋的節點數比 Alpha-Beta 切捨演算法拜訪的節點數多的例子	斥候搜尋時 TEST 拜訪的節點數比 Alpha-Beta 切捨多的例子
p.157 的圖 7.8 之圖標題	斥候演算法拜訪最少的節點數的例子	斥候演算法拜訪最少節點數的例子
p.163 的圖 7.1 之圖標題	圖 7.10 中的函數呼叫過程	依序條例圖 7.10 中的函數呼叫過程
p.165 的 7.4.3 節	在第 14 行中，由於搜尋第一個子節點時，n 的值是 $beta$ 且前一行的 m 目前是 $-\infty$，因此 m 會被設成第一個子節點以 $[alpha, beta]$ 搜尋所得的值。而在之後的節點中，只有當測試成功（$t > m$）時會額外考量另外兩個條件：$detph < 3$ 以及 $t \geq beta$。	在第 14 行中，由於搜尋第一個子節點時，n 的值是 $beta$ 且前一行的 m 目前是 $-\infty$，因此 m 會被設成第一個子節點以 $[alpha, beta]$ 搜尋所得的值。而在之後的節點中，只有當測試成功（$t > m$）時會額外考量另外兩個條件：$depth < 3$ 以及 $t \geq beta$。
p.166 的 7.4.4 節	最後，正反斥候演算法是一個結合斥候演算法與 Alpha-Beta 切捨演算法的混合式演算法。同時也是採用傳統搜尋演算法的程式之現況最佳解（state-of-the-art）。	最後，正反斥候演算法是一個結合斥候演算法與 Alpha-Beta 切捨演算法的混合式演算法，同時也是採用傳統搜尋演算法的程式之現況最佳解（state-of-the-art）。
p.169 的 8.1 節	使用先前搜尋時所累積的資訊，例如將搜尋過的資訊儲存在同形表（transposition table），搜尋過程中，再去檢查同形表中是否已存有相同之搜尋紀錄。	使用先前搜尋時所累積的資訊，例如將搜尋過的資訊儲存在同形表（transposition table），搜尋過程中在去檢查同形表中是否已存有相同之搜尋紀錄。
p.170 的 8.1 節	……當區間愈大，計算時間愈多，但回傳最佳值的機率也愈大，反之計算時間少，回傳最佳解的機率愈少。	……當區間愈大計算時間愈多，但回傳最佳值的機率也愈大；反之，計算時間少，回傳最佳解的機率愈少。
p.171 的 8.2 節	除少數單人對局之對局過程可以用樹的方式展開外，也就是相同節點僅透過一條路徑到達，大部分的對局是以對局圖的模式存在，使得不同的路徑都可以到達同一個節點。透過同形表彙集重複使用搜尋過的資訊，而儲存資訊的資料庫必須足夠大， 才能容納搜尋節點的紀錄數值，伴隨而來的是如何有效率地在資料庫中找尋資料，因此使用了雜湊表作爲設計同形表的資料結構。	除少數對局過程可以用樹的方式展開外，也就是相同節點僅透過一條路徑到達，大部分的對局是以對局圖的模式存在，使得不同的路徑都可以到達同一個節點，此外如第 7.4.2 節之正反斥候演算法也會對同一節點重複搜尋。透過同形表才能彙集重複使用搜尋過的巨量資訊，伴隨而來的是如何有效率地在龐大同形表中找尋資料，因此使用了雜湊表作爲設計同形表的資料結構。
p.172 的 8.2 節	3. 最佳值：搜尋子樹的最佳值，可能爲正確值、剪枝值（$alpha$值和$beta$值）。	3. 最佳值：搜尋子樹的最佳值，可能爲正確值、剪枝值（上界值或下界值）。

p.172 的 8.2 節	演算法實作過程可以有各種不同考量，以下提供一種基本的實作演算法：搜尋盤面紀錄時，要先比對目前之盤面狀態、著手方是否與搜尋紀錄相同，搜尋紀錄的元素包含最佳值 v'、搜尋深度 $depth'$、精確解（exact value）和最佳著手 m'。	演算法實作過程可以有各種不同考量，以下提供一種基本的實作演算法：搜尋盤面紀錄時，要先比對目前之盤面狀態、著手方是否與搜尋紀錄相同，搜尋紀錄的元素包含最佳值 m'、搜尋深度 $depth'$、精確解（exact value）和最佳著手。
p.172 的 8.2 節	其中 *exact* 爲布林值，若搜尋結果是精確解則爲眞（$true$），若是產生Beta剪枝則爲僞（$false$）。假設目前盤面 p 之搜尋深度是 $depth$：	其中 *exact* 爲旗標，代表是精確解、上界或下界。假設目前預計搜尋的深度爲 $depth$：
p.172 的 8.2 節	• 若$depth \leq depth'$，直接回傳 m' 作爲搜尋最佳解。	• 若$depth \leq depth'$，若爲精確解則直接回傳 m' 作爲搜尋最佳解；若是上、下界值，則更動其相對應之 *beta* 或 *alpha* 值。
p.173 的 8.2.2 節標題	加入同形表之正反斥候演算法	加入同形表之Alpha-Beta 切捨演算法
p.173 的 8.2.2 節	Algorithm 28爲使用同形表之正反斥候演算法。	Algorithm 28爲使用同形表之Alpha-Beta切捨演算法的一部分（最大化節點）。
p.173 的 8.2.2 節	當產生Beta剪枝（第 8 ∼ 10 行、第 14 ∼ 16 行）；直接儲存*beta*值至同形表中，當搜尋結果是精確解（第 18 行），直接儲存精確值至同形表中；若產生同形表中獲取資料且盤面紀錄之搜尋深度較淺，則將紀錄優先搜尋。搜尋完後，再更新同形表中之盤面紀錄。	當產生Beta剪枝（第 8 ∼ 10 行、第 14 ∼ 16 行）；直接儲存下界值至同形表中，當搜尋結果是精確解（第 18 行），直接儲存精確值至同形表中；若產生同形表中獲取資料且盤面紀錄之搜尋深度較淺，則將紀錄優先搜尋。搜尋完後，再更新同形表中之盤面紀錄。
p.174 的 Algorithm 28	2 if yes, then HASH HITS, retrieve the stored value m' $depth'$ and *exact* flag;	2 if yes, then HASH HITS, retrieve the stored value m', $depth'$ and *exact*;
p.174 的 Algorithm 28	6 $m := -\infty$ or m' if HASH HITS;	6 $m := -\infty$ or m' if HASH HITS with an exact value;
p.174 的 Algorithm 28	17 **if** $m > beta$ **then**	17 **if** $m > alpha$ **then**
p.174 的 Algorithm 28	20 update it as a lower bound;	20 update it as an upper bound;
p.175 的 8.3.2 節	如：西洋棋有黑、白2種不同的棋子，棋盤上有361個位置，共產生2×361個隨機數，其資料結構爲$s[1\cdots2][1\cdots361]$。	如：圍棋有黑、白2種不同的棋子，棋盤上有361個位置，共產生2×361個隨機數，其資料結構爲$s[1\cdots2][1\cdots361]$。
p.180 的 Algorithm 29	**Algorithm 29:** IDAS(position p, integer $limit$, integer $threshold$)	**Algorithm 29:** IDAS(position p, integer $limit$, value $threshold$)
p.181 的 Algorithm 30	**Algorithm 30:** IDAS′(position p, integer $limit$, integer $threshold$)	**Algorithm 30:** IDAS′(position p, integer $limit$, value $threshold$)
p.186 的 8.5.5 節	殺手捷思法（killer heuristic）是由否議表演變而來，主要是觀察逐層加深儲存的所有主要變化路徑中，相對應深度相同的好走步機率很大。	殺手捷思法（killer heuristic）是由否議表演變而來，主要是觀察逐層加深儲存的所有主要變化路徑中，儲存在相對應深度的好走步有很高機率是一樣的。
p.187 的 8.5.6 節	從否議表記錄 *current_depth_limit* 條主要變化路徑，到殺手捷思法記錄兩條主要變化路徑，而歷史捷思法是當儲存好的著手時，不管深度爲何，由於無法只記錄一個好的著手，因此將所有可能的著手作出現次數的統計，.....	從否議表記錄 *current_depth_limit* 條主要變化路徑，到殺手捷思法記錄兩條主要變化路徑，而歷史捷思法是當儲存好的著手時，不管深度爲何，由於無法只記錄一個好的著手，因此統計所有可能著手出現次數，再參考此統計值做走步選擇。.....

p.192 的 8.7.1 節	主要概念是，當輪我出手時，我方不走，讓對方再走，也就是對方連續下兩步，再評估讓步後的盤面分數。	主要概念是，當輪我方出手時，我方不走，讓對方再走，也就是對方連續下兩步，再評估讓步後的盤面分數。
p.193 的 8.7.1 節	已經在虛手切捨的遞迴呼叫中，不可以再進行虛手切捨，否則可能過度剪枝	已經在虛手切捨的遞迴呼叫中（亦即 in_null 是 $TRUE$），不可以再進行虛手切捨，否則可能過度剪枝
p.194 的 8.7.2 節	目前已正在進行較晚考慮著手之搜尋裁減，不宜再遞迴進行裁剪，以免實際搜尋深度過淺。	目前已正在進行較晚考慮著手之搜尋裁減，不宜再遞迴（亦即 in_lmr 爲 $TRUE$）進行裁剪，以免實際搜尋深度過淺。
p.195 的 Algorithm 36	**Algorithm 36:** F4.4′(position p, value $alpha$, value $beta$, integer $depth$, Boolean do_null)	**Algorithm 36:** F4.4′(position p, value $alpha$, value $beta$, integer $depth$, Boolean in_null)
p.195 的 Algorithm 36	9 If do_null is $false$, then go to Skip;	9 If in_null is $true$ or P is dangerous, then go to Skip;
p.195 的 Algorithm 36	10 $null\_score$:= F4.4′(p', $beta$, $beta+1$, $depth-R-1$, $false$);	10 $null\_score$:= F4.4′(p', $beta$, $beta+1$, $depth-R-1$, $true$);
p.195 的 Algorithm 36	16 m := max{m, G4.4′(p_1, $alpha$, $beta$, $depth-1$, do_null)};	16 m := max{m, G4.4′(p_1, $alpha$, $beta$, $depth-1$, in_null)};
p.197 的 Algorithm 37	**Algorithm 37:** F4.5′(position p, value $alpha$, value $beta$, integer $depth$, Boolean do_lmr)	**Algorithm 37:** F4.5′(position p, value $alpha$, value $beta$, integer $depth$, Boolean in_lmr)
p.197 的 Algorithm 37	12 **if** not(do_lmr) **and** $i \geq K$ **and** $depth > H+3$ **and** p_i is not dangerous	12 **if** not(in_lmr) **and** $i \geq K$ **and** $depth > H+3$ **and** p_i is not dangerous
p.197 的 Algorithm 37	17 $flag$:= $false$;	17 $flag$:= in_lmr;
p.197 的 Algorithm 37	24 m:=G4.5′($p_i, t, beta, depth-1, do\_lmr$); // re-search	24 m:=G4.5′($p_i, t, beta, depth-1, in\_lmr$); // re-search
p.199 的 Algorithm 38	9 **if** p_1 is a capturing move **then**	9 **if** p_1 is dangerous **then**
p.199 的 Algorithm 38	17 **if** p_i is a capturing move, ... **then**	17 **if** p_i is dangerous **then**
p.213 的附註4	隨機對局產生時，需注意不可走出自填眞眼的走步，因此選取之對局不是眞正隨機，而是幾近隨機（almost random）。	隨機對局產生時，需注意不可走出自填眞眼的走步，因此選取之對局不是眞正隨機，而是幾近隨機。
p.215 的 9.4.2 節	若我們在目前玩的 t 次裡，玩 a 的次數爲 $N_t(a)$，則若 a 愈大，對 a 眞實值的估計值不確定性就會愈低，即 $N_t(a)$ 愈高，$U_t(a)$ 就會愈低。反之，若玩的 a 次數愈少，則對 a 眞實值的估計值不確定性就會愈高，即 $N_t(a)$ 愈低，$U_t(a)$ 就會愈高。	若我們在目前玩的 t 次裡，玩 a 的次數爲 $N_t(a)$，則若 $N_t(a)$ 愈大，對 a 眞實值的估計值不確定性就會愈低，即 $N_t(a)$ 愈大，$U_t(a)$ 就會愈小。反之，若玩的 a 次數愈少，則對 a 眞實值的估計值不確定性就會愈高，即 $N_t(a)$ 愈小，$U_t(a)$ 就會愈大。
p.216 的 9.4.2 節	$\mathbb{P}[Q(a) \geq \hat{Q}_t(a)+U_t(a)] \leq e_t^{-2N(a)U_t(a)^2} \leq p$	$\mathbb{P}[Q(a) \geq \hat{Q}_t(a)+U_t(a)] \leq e_t^{-2N(a)U_t(a)^2} \leq p$
p.219 的 Algorithm 43	8 Update the UCB score of p^*;	8 Update the UCB score of p^* as well as other nodes;
p.220 的 9.5.1 節	接下來，以主要變化路徑的葉節作爲起始局面，	接下來，以主要變化路徑的葉節點作爲起始局面，

p.222 的圖 9.13		
p.222 的 Algorithm 44	**10 Backward propagation:**	**10 Back propagation:**
p.225 的 Algorithm 45	**Algorithm 45:** UCT	**Algorithm 45:** MCTS
p.225 的 Algorithm 45	**14 Backward propagation:**	**14 Back propagation:**
p.229 的 10.2 節	信賴上界的樹搜尋演算法於執行時，會產生一些資訊，...	蒙地卡羅樹搜尋演算法於執行時，會產生一些資訊，...
p.229 的頁尾		在此提醒讀者，本節（第10.2節）將要提及之各項技巧的研發在時序上早於9.6節（信賴上屆的樹搜尋），可以選擇是否和9.6節合併或單獨使用。
p.231 的 10.3.2 節	等價標準差 σ_e 愈大的話，則會有愈多棋步被剪枝，亦即是存在愈少統計上等價的棋步，此時蒙地卡羅樹搜尋演算法的效能愈好，需要愈少的計算時間收斂，但收斂品質可能較差。	等價標準差 σ_e 愈小的話，則會有愈少棋步被剪枝，亦即是存在愈少統計上等價的棋步，此時蒙地卡羅樹搜尋演算法的效能愈好，需要愈少的計算時間收斂，但收斂品質可能較差。
p.253 的 10.3.2 節	接下來，我們會以 AlphaGo 使用卷積神經網路加強蒙地卡羅樹搜尋擴展階段的方法爲中心，來說明在擴展階段結合離線學習知識的方法。	接下來，我們會以圍棋程式如何使用卷積神經網路加強蒙地卡羅樹搜尋擴展階段的方法爲中心，來說明在擴展階段結合離線學習知識的方法。
p.256 的 10.3.2.2 節	同時，我們也用同樣的資料一前面章節所用提到的 BT 模型建立模擬階段用的模擬策略模型 p_π。	同時，我們也用同樣的資料一前面章節所用提到的 BT 模型建立模擬階段用的模擬策略網路（rollout policy network）模型 p_π。
p.308 的 13.2 節	在圖 13.3 中，假設對局規定走出重複盤面的一方爲輸。則 $A \to B \to E \to I \to J \to H \to E$ 會因爲棋規而導致輸棋，因此會在同形表中被記錄 H 盤面爲黑方輸。若接下來搜尋路徑爲 $A \to B \to D$ 同樣也是輸棋。$A \to C \to F \to H$ 由於 H 被記錄爲輸棋，因此也是輸棋。由此兩條路徑可以推導出 A 盤面是白方輸棋，但事實上 $A \to C \to F \to H \to E \to G$ 乃是 A 點爲白方可贏棋的盤面。	在圖 13.3 中，假設對局規定走出重複盤面的一方爲勝。則 $A \to B \to D \to G \to I \to J \to D$ 會因爲棋規而導致輸棋，因此會在同形表中被記錄 J 盤面爲黑方勝。若接下來搜尋路徑爲 $A \to B \to D \to H$ 則是贏棋路徑。$A \to C \to F \to J$ 由於 J 被記錄爲輸棋，因此也是輸棋。由此兩條路徑可以推導出 A 盤面是白方輸棋，但事實上 $A \to C \to F \to J \to D \to H$ 乃是 A 點爲白方可贏棋的盤面，其原因在 J 被錯誤判斷爲輸棋盤面。
p.310 的 圖 13.3		

p.313 的 13.4	Algorithm 59 隨機行爲節點的搜尋順序很重要，這基本上和找好的著手次序是相同的問題。如果能先搜尋可以大幅縮短區間$[m_i, M_i]$的可能性，則很容易加速Alpha-Beta 切捨的搜尋速率。此外，本書介紹之 F2.1′ 是修改自第 6 章硬式失敗版本的 F2′，如果使用軟式失敗版本的 F3′，甚至是第 7 章正反斥候（F4′）則需考量回傳值若超出上下界限時如何算出合理的期望值，這是值得研究的進階課題。	Algorithm 59 隨機行爲節點的搜尋順序很重要，這基本上和找好的著手次序是相同的問題。如果能先搜尋可以大幅縮短區間$[m_i, M_i]$的可能性，則很容易加速Alpha-Beta 切捨的搜尋速率。此外，本書介紹之 F3.1′ 是修改自第 6 章軟式失敗版本的 F3′，如果使用硬式失敗版本的 F2′，甚至是第 7 章正反斥候（F4′）則需考量回傳值若超出上下界限時如何算出合理的期望值，這是值得研究的進階課題。
p.314 的 Algorithm 59	**Algorithm 59:** F2.1′(position p, value $alpha$, value $beta$)	**Algorithm 59:** F3.1′(position p, value $alpha$, value $beta$)
p.314 的 Algorithm 59	5 $m := alpha$;	5 $m := -\infty$;
p.314 的 Algorithm 59	8 $t :=$ Star1_F2.1′$(p_i, n, m, beta)$;	8 $t :=$ Star1_F3.1′$(p_i, n, \max\{alpha, m\}, beta)$;
p.314 的 Algorithm 59	10 $t :=$ G2.1′$(p_i, m, beta)$;	10 $t :=$ G3.1′$(p_i, \max\{alpha, m\}, beta)$;
p.315 的 Algorithm 60	**Algorithm 60:** Star1_F2.1′(position p, node n, value $alpha$, value $beta$)	**Algorithm 60:** Star1_F3.1′(position p, node n, value $alpha$, value $beta$)
p.315 的 Algorithm 60	2 $A_0 := c(alpha - v_{max}) + v_{max}$, $B_0 := c(beta - v_{min}) + v_{min}$;	2 $A_0 := c(alpha - v_{max}) + v_{max}$, $B_0 := c(beta - v_{min}) + v_{min}$;
p.315 的 Algorithm 60	7 $t :=$ G2.1′$(p_i, max\{A_{i-1}, v_{min}\}, min\{B_{i-1}, v_{max}\})$;	7 $t :=$ G3.1′$(p_i, max\{A_{i-1}, v_{min}\}, min\{B_{i-1}, v_{max}\})$;
p.320 的 Algorithm 61	22 Prove u or disprove u;	22 Solve u;
p.320 的 Algorithm 61	22 Prove u or disprove u;	22 Solve u;