國立交通大學

多媒體工程研究所

碩士論文

將蒙地卡羅樹狀搜尋應用於《星海爭霸》中的戰術決策

Applying Monte Carlo Tree Search for Tactical Decision-making in StarCraft

研 究 生：蔣承翰

指導教授：孫春在　教授

中 華 民 國 一 〇 五 年 七 月

將蒙地卡羅樹狀搜尋應用於《星海爭霸》中的戰術決策

Applying Monte Carlo Tree Search for Tactical Decision-making in StarCraft

研 究 生：蔣承翰　　Student: Cheng-Han Chiang

指導教授：孫春在　教授　　Advisor: Chuen-Tsai Sun

國立交通大學

多媒體工程研究所

碩士論文

A Thesis

Submitted to Institute of Multimedia Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

In

Computer Science

July 2016

Hsinchu, Taiwan

中華民國一〇五年七月

將蒙地卡羅樹狀搜尋應用於《星海爭霸》中的戰術決策

學生：蔣承翰　　指導教授：孫春在　教授

國立交通大學多媒體工程研究所碩士班

摘　要

本研究將以 Monte Carlo tree search 來處理即時戰略遊戲（real-time strategy）中的 tactical decision-making，也就是操作一群單位和敵方的單位作戰。選定的遊戲為 StarCraft，其在即時戰略遊戲中的地位，有如西洋棋在桌遊中的地位。之所以想嘗試 MCTS 在這個問題上的效果，是因為 MCTS 能夠以較少的專家知識，在龐大的狀態中使用類似統計採樣的方法，來找到好的選擇；同時，即時戰略遊戲強調即時性，這也讓能夠隨時停止搜尋的 MCTS 顯得更為適用。

藉由 UCT considering duration，這個因為加入了 durative action 而能夠處理即時遊戲的演算法，我們能夠推展出一套遊戲抽象化的方式，來做出一個供 MCTS 使用的模擬器。實驗首先找到 MCTS 適合的 exploration constant 和最大子節點數量，再分別交叉測試了幾種單位數量和單位種類，分別為 4/8/16/32/50 個單位，以及純粹遠距單位/純粹近戰單位/混合單位。

實驗結果顯示，在 evaluation function 中，表現最好的是 HPD，一個定義為雙方總生命值差值的函式，而非其他有加入傷害資訊的函式。加入更多單位資訊我認為是對的，但大概是加入的方式不對，效果才不盡理想。另外，MCTS 在遠距單位上的表現比較好，對抗內建 AI 的勝率平均大於 80%，而在近戰單位上的表現則是極差，勝率只有 10-20%，很可能是因為，模擬器為了效能的顧慮，沒有考慮單位碰撞，而單位碰撞對於近戰策略的擬定至關重要，無法用這種方式估計。MCTS 在單位過多的時候，效果也會大打折扣。

這個結果對其他研究者來說，已經算是揭露了 MCTS 在即時戰略上的可能性，如果他們有更強大的運算資源，我想肯定會有更好的表現。對遊戲開發者來說，這個結果代表，他們可以藉著使用 MCTS 在自家開發的即時戰略遊戲上，來做出強度更高、更有趣的電腦玩家。

Applying Monte Carlo Tree Search for Tactical Decision-making in StarCraft

Student: Cheng-Han Chiang Advisor: Chuen-Tsai Sun

Institute of Multimedia Engineering
National Chiao Tung University

ABSTRACT

In this thesis, we apply Monte Carlo tree search for tactical decision-making in StarCraft, which is about controlling units to combat with opponent's units in real-time strategy games. MCTS can use less expert knowledge to achieve high performance and adaptive plays. And, for real-time games, MCTS can stop searching at any time to return the current best move.

Applying UCT considering duration, which considers durative moves in UCT, we can implement a simulator for MCTS with reasonable game abstraction. In experiments, we found the value for exploration constant and the number of max children for nodes in tree first. Then we tested different number of units and different kinds of units, including 4/8/16/32/50 units and pure ranged/pure melee/mixed units' compositions.

First, the results showed that the best evaluation function is HPD, which was defined as the difference of total hitpoints of both sides, rather than other functions considering units' damage. It's not wrong to consider more information of units, but perhaps it's suitable to use them that way. Second, MCTS did more well on ranged units than melee units. Because we needed better performance of simulator, ignoring unit collisions seemed to prevent it from finding good strategy for melee units. And third, MCTS couldn't handle too much units.

This research prevails the applicability of MCTS to real-time strategy. If other researchers can afford more computational resources, it is certain that this method would show better performance and results. On the other hand, for game developers, they can use this method to implement a more interesting and better built-in AI of their RTS games.

誌　　謝

我有一個很奇怪的指導教授。

孫老師幾乎從來不向學生追討什麼，也不指定研究題目，不強迫人接續過往學長姊的研究，該讀的論文也請自己花時間，不會每個星期逼你要報一篇論文，進度沒有太多限制。這樣的研究所生涯，老實說跟我想像的完全不一樣，也跟朋友分享的經驗一點都不像。就在朋友抱怨著當初指導老師是如何騙他進去的時候，我真的覺得好幸運，能給孫老師教到。

老師給我的，不是一大堆以後不知道有什麼用的艱深知識，而是足夠的自主空間，自己決定我要什麼。我花了滿長時間才了解到，老師施行的或許就是他常說的，無為而治。無為而治不是什麼都不做，而是依循自然之理，順應事物本身的品性，一切終將自主性的歸於定位，所以才看起來像是什麼都沒做。老師不直接告訴人該怎麼做，或是該做什麼，可是在需要的時候，老師就會花很長的時間，給予很多建議與鼓勵，引導我做出選擇，無論是對研究還是生涯規劃。如果不是老師，我今天大概沒辦法做這個題目，也留不住什麼課餘時間，拿來從事我喜歡的寫作吧。

我在老師身上也看到了真正的謙遜與開明。老師對於知識似乎存有很純粹的好奇心，以致於任何新的事物，老師都希望能夠有機會了解，也從不避諱於向我們請教。老師常說，「那些對陌生的東西，還沒試就直接否定的人，其實是自己放棄了了解其中樂趣的機會。」另外，老師也說過，他不會像一些長輩很喜歡給晚輩建議，甚至不太敢給，因為心裡明白，自己生長的時代，和現在這個時代，已經完全不一樣了。實在好難得，一個教授願意這樣向學生坦承。但我也必須說：別擔心，老師你一直都在跟上時代。

如今，我的口試順利通過了，感謝孫老師兩年來的盡心指導；感謝胡毓志老師，給我人工智慧相關的專業指點；感謝林珊如老師，這麼仔細的閱讀我的論文，替我點出許多應該修改的部分；感謝一起口試的夥伴，柏皓和沛亘，以及來幫忙的冠霆，以我當天的狀況，口試以外的事情都不太能思考，沒有你們的話，口試一定沒辦法這麼順利。

目　　錄

表 目 錄

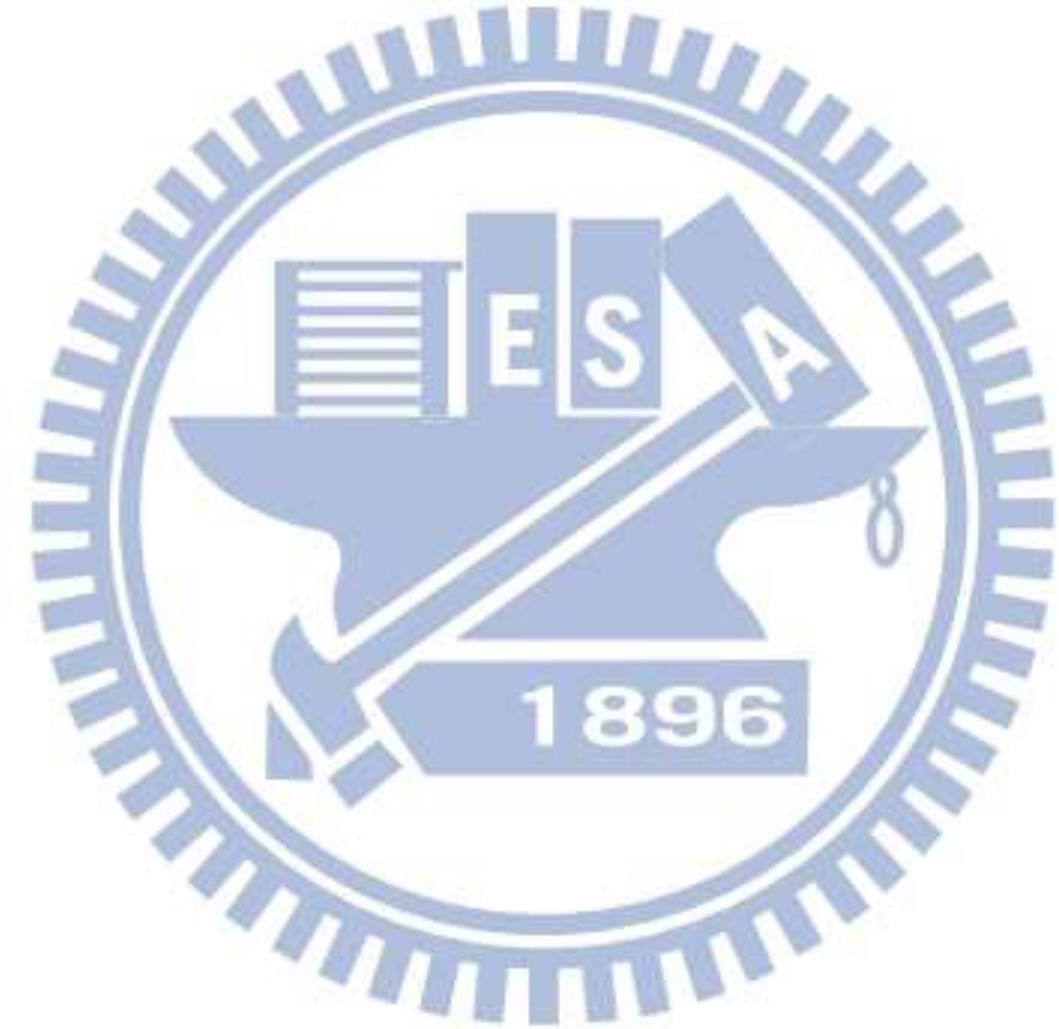


圖 目 錄

一、 緒 論

1.1 研究動機

對人工智慧的相關研究來說，遊戲是一個非常適合的場域，包括各式桌上遊戲（board games）與數位遊戲（video games），例如西洋雙陸棋 backgammon (Tesauro 1995)、橋牌 bridge (Ginsberg 1999)、撲克 poker (Billings, Castillo et al. 1999) 和圍棋 *Go* (Bouzy and Helmstetter 2004)，而數位遊戲則有 *StarCraft*[1]、*Magic: The Gathering*[2] (Cowling, Ward et al. 2012) 等等，許多人工智慧的相關技術，在以上這些遊戲中都有長足的發展。相對於複雜而不斷變動的現實世界，遊戲提供了一套具有固定規則和目標的挑戰，讓人工智慧能在應用於現實生活中之前，先在一個模擬環境裡發展(Schaeffer 2001)。

而數位遊戲中，即時戰略遊戲是策略遊戲（strategy games）的一種，大略來說玩法上就像是一個簡化的戰爭模擬器，玩家需要維繫己方的經濟及單位生產，目標是破壞對方所有建築物即為勝利。在這類遊戲中通常充滿著許多會讓兵種或戰術相剋的遊戲機制，換句話說，如果沒辦法及時針對敵方的戰術做出適當的反應，一旦交戰起來幾乎可以先行宣告失敗了。因為這個特點，無調適性的人工智慧在遊戲中更顯劣勢。相較於人類玩家能做出的戰術規劃，這類遊戲的人工智慧在如此龐大的狀態空間（state space）中，如果僅僅以腳本或有限狀態機等等無調適性的技術，要直接考慮所有可能性是不可行的(Robertson and Watson 2014)。

要解決這個問題，許多學者試著將具有調適性的人工智慧技術應用在即時戰略遊戲中，盡可能使用較少的專家知識（expert knowledge）去達到較高的強度，例如 reinforcement learning (Shantia, Begue et al. 2011)、neuroevolution (Gabriel, Negru et al. 2012)、Bayesian model (Synnaeve and Bessiere 2011) 及 Monte Carlo tree searching (Chung, Buro et al. 2005, Sailer, Buro et al. 2007, Balla and Fern 2009, Zhe, Nguyen et al. 2012)。其中，Monte Carlo tree searching 利用大量機率採樣（stochastic samples），於龐大的狀態空間中得到統計結果（statistical outcome），這樣的方式已經在許多遊戲中被證實為可行而且有效，例如橋牌 bridge (Ginsberg 1999)、撲克 poker (Billings, Castillo et al. 1999) 以及自從應用 Monte Carlo 之後產生重大突破的圍棋 *Go* (Bouzy and Helmstetter 2004)。

對即時戰略遊戲來說，Monte Carlo tree searching 正是一個非常適合的人工智慧技術，它對於處理這類遊戲中的不確定性（uncertainty）、隨機性（randomness）、龐大決策空間（decision spaces）和對手行動（opponent actions）非常有效(Robertson and Watson 2014)。然而，以往將 Monte Carlo tree searching 應用在即時戰略遊戲中單位控制的研究

[1] Blizzard Entertainment: *StarCraft*: http://us.blizzard.com/en-us/games/sc/
[2] *Magic: The Gathering* Official Site: http://magic.wizards.com/

並不多，就算有也大多是在規則相對簡化的開源即時戰略遊戲或引擎上做研究，像是 *ORTS*[3] (Chung, Buro et al. 2005) 和 *Wargus*[4] (Balla and Fern 2009)。在各式人工智慧技術相繼在 *StarCraft* 上做驗證的熱潮相映之下，Monte Carlo tree searching 的缺席也更顯得特異，原因之一可能是，在 *StarCraft* 上始終缺乏一個有效的模擬器(simulator)(Robertson and Watson 2014)，因為它畢竟是商業遊戲，研究者無法取得遊戲程式內的所有規則數值，而 Monte Carlo tree searching 卻又特別倚賴模擬器去做採樣的原故。

綜合以上所述，Monte Carlo tree searching 正符合即時戰略遊戲對人工智慧的需求，既能有效降低對腳本及專家知識的依賴，減少這方面的開發成本 (Chung, Buro et al. 2005)，同時又可以因為此方法的調適性，讓它能夠應付更多未知的情況。本次研究將先著眼於處理 *StarCraft* 的 micromanagement，也就是兩軍間作戰時的單位控制（units control），提出一套能在商業遊戲上做驗證的 Monte Carlo tree searching 方法，以及其適用的模擬器。

1.2 研究背景

1.2.1 即時戰略遊戲（RTS, real-time strategy）

即時戰略遊戲是策略遊戲（strategy games）的一個分支，規則上就像是一個簡化的戰爭模擬器。這類遊戲強調即時制（real-time），也就是說，所有玩家能夠同時各自進行動作，試著在相同時間中盡可能有效的下達最多指令以獲得最終勝利，和那些讓玩家輪流行動的回合制（turn-based）遊戲恰好相反。

即時戰略遊戲通常包含了資源收集（resource collection）、建築物建造（building constructions）、科技升級（technology upgrades）、單位生產（units production）及單位控制（units control）等等要素，玩家在遊戲裡必須在有限的注意力下，於眾多繁雜的事務中做好平衡，並且規劃好行動順序，挑選當下最重要的事情來做，同時又要根據對手的狀況來及時調整戰術，一切就為了能在各個關鍵的時間點，讓對的單位組合在最適當的地點就位 (Robertson and Watson 2014)，去完成階段性的戰略目標，一直到破壞掉對方所有的建築物，贏得最後勝利。

那些專注於長程目標（long-term goal）的高階規劃（high-level planning）以及決策（decision-making）的概念，在遊戲社群中時常被稱為 macro（macromanagement），而這裡套用 Robertson 和 Watson 於 2014 年所發表的分類方式來稱之為 strategic decision-making；相對的，另一些以短程目標（short-term goal）為主的相關決策，例如兩軍短兵相接（combat）時的單位指揮，社群中稱為 micro（micromanagement），以那套分類方式來說則是 tactical decision-making (Robertson and Watson 2014)。若替前面提

[3] *ORTS*: https://skatgame.net/mburo/orts/
[4] *Wargus*: http://wargus.sourceforge.net/index.shtml

過的那些即時戰略遊戲的要素做歸類的話，資源收集、建築物建造、科技升級和單位生產是屬於 strategic decision-making，而只有單位控制會被歸於 tactical decision-making。在本次研究中，會先將實驗的範圍界定在 tactical decision-making 內。

大多的即時戰略遊戲為了要加入偵查（scouting）元素，讓對戰雙方不知道對方目前的完整情況，會有戰霧（fog of war）的設定，也就是沒有自身單位存在的地方，就沒有那區的視野（field of view），也就沒辦法知道那個位置附近的目前資訊。這讓遊戲中的不確定性（uncertainty）分成了兩種：intentional 及 extensional。intentional 的不確定性來源是已經在視野內的非己方單位，而 extensional 的不確定性則是來自沒有視野的區域，例如對方基地的戰略配置，或突如其來的增援(Synnaeve and Bessiere 2011)。這次研究只先處理 intentional 的部分，也就是參與戰鬥的單位始終都只有視野內的這些。

Strategic decision-making (Macromanagement)	資源收集（resource collection）、建築物建造（building structures）、科技升級（technology upgrades）、單位生產（units production）、偵查（scouting）
Tactical decision-making (Micromanagement)	單位控制（units control）

表格 1：即時戰略遊戲要素的決策類型 (Robertson and Watson 2014)。

在如此複雜的遊戲規則底下，即使只將 tactical decision-making 個別挑出來看，對每個單位來說，如果暫且將環境視為離散空間（discretized space）的話，它有 24 個選擇可以移動（詳見圖 1），外加停留和攻擊，有些單位還能施放技能，每個幀（frame）就至少有 26 種選擇，更不用說實際遊玩時，場上往往都有超過 20 個單位需要操作(Synnaeve and Bessiere 2011)，而這都還沒有討論到 strategic decision-making 部分的各種可能性，如建築物建造順序和位置等等。因此，即時戰略遊戲的狀態空間（state space）實在非常龐大，只靠一般的搜尋演算法是不可行的(Robertson and Watson 2014)。

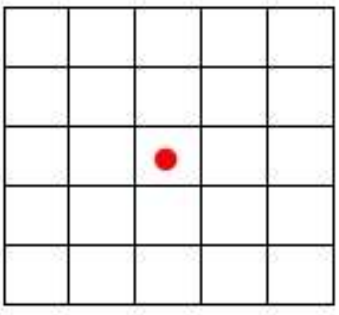


圖 1：若將環境視為離散空間的 24 個移動選擇。紅點為單位所在位置。

由人工智慧的發展進程來看，最先被破解的，一個具有代表性的桌上遊戲是象棋，再來就是日前才剛被 Google 的 AlphaGo 以 4：1 戰勝職業九段棋士的圍棋，兩者皆屬於 fully observable 的回合制遊戲，只是後者的複雜度又高上許多。從象棋到圍棋，代表的是人工智慧對於賽局中複雜度的應付能力。下一步，Google 已經宣布要挑戰的，就是這款即時戰略遊戲 *StarCraft*。[5]這個發展所代表的，已經不只是遊戲複雜度的提升，還同時包括了針對隱藏訊息（戰霧）的處理，以及如何在即時制下，用更短的時間擬訂出策略，並給出有效的指令。

1.2.2 StarCraft: Brood War

StarCraft 是一款在即時戰略遊戲中具有代表性的作品，它在即時戰略遊戲中的價值，就有如西洋棋在桌上遊戲中的地位(Robertson and Watson 2014)。根據 2011 年的統計資料，這款遊戲在世界上總共銷售了超過 1 億套，獲得 50 多個來自遊戲產業的獎項，其中有超過 20 個獎項是「年度最佳遊戲（game of the year）」，每年各國以這款遊戲舉辦的相關競賽總獎金超過幾百萬美金，吸引了來自世界各地的職業電競選手爭相參加(Churchill and Buro 2011)。時至 2015 年的今日，即使續作 *StarCraft 2*[6]推出已久，這款前作的比賽卻沒有因此消失，反而還有一群死忠的玩家、選手和觀眾持續關注。*StarCraft: Brood War* 則是它的一個資料片（expansion）。

這款遊戲提供了三個截然不同，卻又十分平衡（balanced）的種族供玩家選擇，每個種族各自有其特性，能夠利用優勢發展出不同的戰術，而種族之間的各種對戰組合，更是將整個遊戲的複雜度提升到另一個層次。先前所述的即時戰略遊戲要素它也都有，包括戰霧和視野等等。它的幀數（frames）固定在 24 fps，也就是一秒有 24 個幀。

和 *StarCraft* 相關的人工智慧研究之所以會如此盛行，還有一個很重要的原因是，網路上有一套由第三方開發的函式庫 *BWAPI*[7]供大家使用；同時因為有了這樣的工具，以這款遊戲為主題的人工智慧競賽也蓬勃發展，如 AIIDE[8]和 CIG[9]所舉辦的比賽。*BWAPI* 是一個以 C++開發的框架（framework），完全免費而且開源，讓程式能夠直接和 *StarCraft* 做互動，包括取得建築物和單位的狀態（status），以及建造建築物、執行各項升級或移動單位。*BWAPI* 在預設情況下是無法作弊的，程式所能獲得的資訊，就如同人類玩家坐在電腦前遊玩能取得的相同，並不會因此能看穿戰霧背後隱藏的敵方單位；在程式運行中，預設也會關閉使用者輸入（user input），讓人不能從旁影響結果，確保了比賽的公平性。而本次研究所需的程式，同樣也會採用 *BWAPI* 來開發。

---

[5] StarCraft Could Be The Next Frontier For Google's AlphaGO AI | Kotaku Australia
http://www.kotaku.com.au/2016/03/starcraft-could-be-the-next-frontier-for-googles-alphago-ai/

[6] *StarCraft 2* Official Site: http://www.starcraft2.com

[7] BWAPI: http://bwapi.github.io/

[8] AIIDE *StarCraft* AI Competition: http://www.starcraftaicompetition.com

[9] CIG *StarCraft* AI Competition: http://cilab.sejong.ac.kr/sc_competition/

圖 2：才剛初始一場遊戲沒多久的 *StarCraft* 畫面。位於圖片正中央的是一個指揮中心，從底部界面中央區域可以看出，它正在生產一個 SCV；圖片右邊有縱向一整排的藍色礦物，是遊戲中主要的經濟來源之一，從指揮中心到礦物之間有許多 SCV 正在來回採集著；右上角列出的則是目前已經採集的各類資源量。

圖 3：部隊離開基地之後的 *StarCraft* 遊戲畫面。位於圖片左下角的介面是小地圖，以俯瞰的方式綜觀整個地圖，明亮的部分是有視野的區域，黑暗的部分則是受戰霧隱藏，失去視野，其中的綠色方塊指的是己方的單位和建築物，白色方框是目前畫面所在區域的可視範圍；圖片上頭有兩個紅色的己方陸戰隊；右上方的黑色區域受戰霧隱藏著。

1.2.3 Monte Carlo Tree Search

在機率性的（stochastic）和不完全資訊的（imperfect information）對戰遊戲中，當玩家的選擇非常多，要憑藉著單純的搜尋演算法做 adversarial planning 是不可行的，而 Monte Carlo tree searching 正好非常適合處理這類問題(Chung, Buro et al. 2005)。它並不是直接去搜尋最佳解，而是利用大量機率採樣（stochastic samples）的方式，於龐大的狀態空間（state space）中得到統計結果（statistical outcome）。也因為如此，Monte Carlo tree searching 不需要太多的專家知識（expert knowledge），也能自行發展出有效的結果。統計的過程可以根據時間限制需求隨時中止，依舊能得出一個到當下為止最好的選擇。這樣的方式已經在許多遊戲中被證實為可行而且有效，例如橋牌 bridge (Ginsberg 1999)、撲克 poker (Billings, Castillo et al. 1999) 以及圍棋 *Go* (Bouzy and Helmstetter 2004)。

Monte Carlo tree searching 的基本執行程序為：首先選擇（selection）一個新的行為（action），用極少的 heuristic 非常快速的將遊戲玩到最後（simulation），再將勝負結果（rollout）回傳給在這之前做過的行為程序，然後不斷重複這個過程，直到時間到達限制，便以目前的統計結果，選擇出當下最好的行為來下決策。

1.3 研究問題

此次研究主要目的為，使用 Monte Carlo tree searching 開發一套可以用於 *StarCraft: Brood War* 的人工智慧方法，來驗證 MCTS 用於即時戰略遊戲中 tactical decision-making 的可能性。其中包括設計一個有效的遊戲模擬器（simulator）以供統計採樣（sampling），來處理交戰中的 tactical decision-making，也就是操作單位（units control）戰勝對方。過程中不考慮視野（field of view）以外的不確定性（uncertainty）以及高階規劃（high-level）相關的 strategic decision-making，參與過程的單位始終都會是同一群，不會額外增援。

StarCraft: Brood War 遊戲本身已經有設計人工智慧，難度沒有變化，只有一種。研究結果會藉由和內建的人工智慧，在額外製作的自訂地圖上對戰來做驗證，盡可能使用較少的專家知識，來探討適合的 MCTS 參數設定及 evaluation function 的設計，分析在不同單位數量及單位種類上的效果。

二、 文獻探討

2.1 即時戰略遊戲

比較早期的人工智慧研究，都聚焦於比較傳統的桌上遊戲（board games），例如西洋棋和圍棋；到了 2001 年，Laird 和 van Lent 在一篇期刊論文中提出，電子遊戲（video games）是一個非常適合的人工智慧研究場域(Laird and VanLent 2001)，這才引起了其他學者開始重視起這個領域的潛力。而緊接著在 2003 年，Buro 和 Michael 更進一步提出了幾個論點來說明，為什麼即時戰略遊戲所建構起的複雜問題，對人工智慧來說，是一個值得研究的領域(Buro 2003)。

即時戰略遊戲強調即時性；也就是說，在人的反射神經等等物理條件允許下，一個玩家在執行動作的時候，敵對玩家也同時在下指令。這讓人工智慧在下決策時，並沒有足夠時間去計算出所有可能發生的狀況。另一方面，即時戰略遊戲包含了許多要素，包含了資源收集（resource collection）、建築物建造（building constructions）、科技升級（technology upgrades）、單位生產（units production）及單位控制（units control）等等，另外還要再加上戰霧（fog of war）的設定，玩家無法獲知沒有視野的區域，正在發生什麼事情，這些都讓這個遊戲的狀態空間（state space）變得非常龐大，要遍尋所有可能性是不可行的（infeasible）(Laird and VanLent 2001, Buro 2003)。

許多學者專家為了解決如此複雜的問題，紛紛引進了在其他人工智慧領域已經被證實可行的方法，試圖在即時戰略遊戲中找到相應的解方。因為即時戰略遊戲的要素實在太多太複雜，這些研究大多都只專注在某個部份的問題，只有少數研究是在各個模組的整合上下工夫。以下就引用 Robertson 所提出的分類方式（詳見表格 1）(Robertson and Watson 2014)，分別介紹在即時戰略遊戲上，目前已經被提出來的人工智慧研究。

2.1.1 Strategic Decision-Making

Strategic decision-making 指那些專注於長程目標（long-term goal）的高階規劃（high-level planning）以及決策（decision-making）的概念，在遊戲社群中時常被稱為 macro（macromanagement）。對這類決策來說，一個最主要的挑戰是，必須在資訊不完整（incomplete information）的情況下，擬訂出幾套需要同時配合進行，卻又不一定有階層關係的一連串行為與目標（例如：建築物的建造順序、科技升級的順序、擴建基地的時間點），而且還要根據偵查（scout）到的對手行為，辨識出相應策略，並即時修正己方的策略來反制(Robertson and Watson 2014)。

(1) Case-Based Planning

CBP 是一個會藉由比較眼前情況與過去的類似經驗，來推算出一些可行策略的技術。這些策略都是由一些已經抽象化的計劃（plan）或是子計劃（sub-plan）所組成，而計劃又是以一連串腳本化（scripted）的行為（action）來表示。這類系統有個最主要的問題是，它在策略層次（strategic level）上的反應不夠靈敏，在行為層次（action level）上又顯得反應過度。有時一直要到某個行為出錯了，才對敵方策略有所反應；也有時也只因為某個行為失敗，就放棄掉整個目前計劃，即使它可能還適用(Palma, Sánchez-Ruiz et al. 2011)。

早期的 CBP 相關研究，先嘗試如何將 RTS 中的要素抽象化，以定義出適合的 state、action 和 goal 等等(Aha, Molineaux et al. 2005)。後來則是有更多進一步的研究，在探討如何使用已經人工標註（human-annotated）過的記錄檔，來增進學習（learning）的效果(Ontañón, Mishra et al. 2007, Mishra, Ontañón et al. 2008)。也有研究試圖引進模糊理論（fuzzy），希望能在策略的辨識和選擇上有所進步(Cadena and Garrido 2011)。

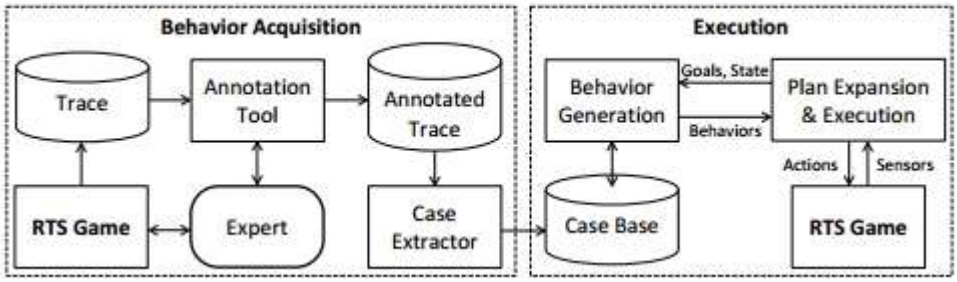


圖 4：一套 case-based planning 的架構，由 Ontañón 等人於 2007 年提出。

(2) Hierarchical Planning

有些研究則是嘗試階層化所有目標，來簡化問題，從最上層的「獲得勝利」開始，再陸續分支下來，一直到最低階的某個特定行為。其中，有個被定義得很好的架構是 Hierarchical Task Network (HTN)，由事項（task）組成，包含順序（ordering）和能夠完成事項的方法（method）。複雜的 task 可以被打散成簡單的 task，直到能夠完全使用遊戲中的行為表示為止(Muñoz-Avila and Aha 2004)。另外也有研究是採用 A Behavior Language，它會用平行式（parallel）、序列式（sequential）和條件式（conditional）來區分這些行為（behavior），以建成一棵樹(Weber, Mawhorter et al. 2010)。

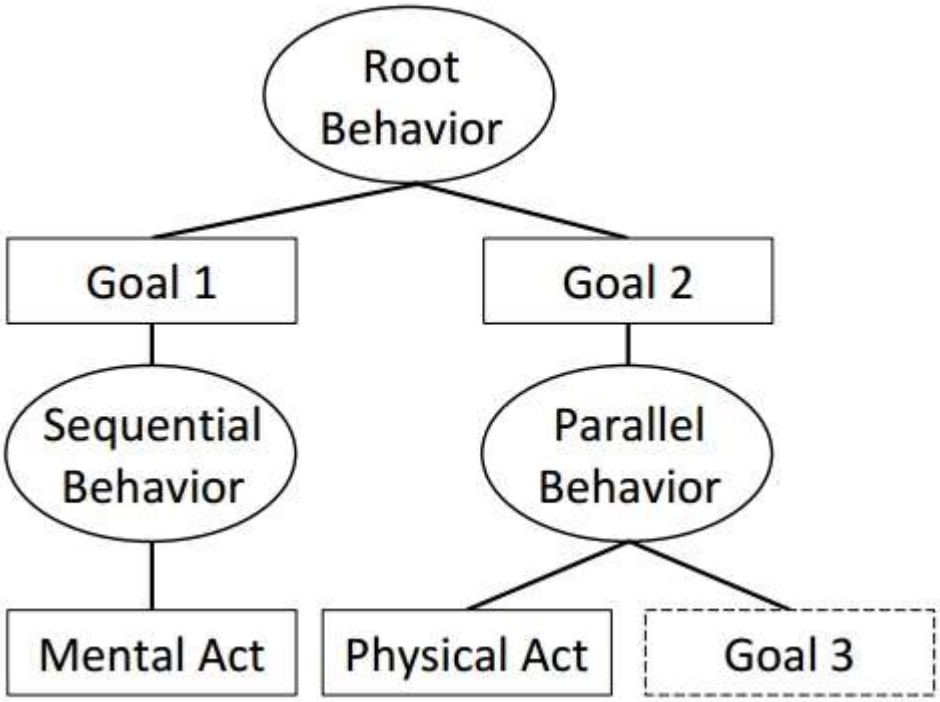


圖 5：一個以 A Behavior Language 建成的樹(Weber, Mawhorter et al. 2010)。

(3) Behavior Trees

Behavior Trees 是相對於腳本（script）來說，更進階的一種人工智慧的設計方法。它由 composite、decorator 和 leaf 等節點(node)組成，而 composite 又可以分為 sequence 和 selector 等，分別會以不同規則去執行子節點中的 behavior。由於這種結構比腳本彈性，又很適合以視覺化工具（visual tools）輔助編輯，對非程式背景的設計師來說更容易上手，在業界廣為使用(Palma, Sánchez-Ruiz et al. 2011)。

Palma 等人於 2011 年的研究就使用了這個方法。他們先藉由機器學習（machine learning）建構出 behavior tree，再讓專家去手動調整這棵樹的各個細部，成功結合了學習（learning）和專家知識（expert-knowledge）(Palma, Sánchez-Ruiz et al. 2011)。

(4) Goal-Driven Autonomy

GDA 主要的運作流程是：以一個 goal manager 產生接下來的目標，交由 planner 做規劃，並且形成一個預期結果，執行 plan，再與預期結果做比較，以不同的部分為依據，調整目標，由 goal manager 繼續產生心的目標，如此往復。(詳見圖 6）這個系統能夠很有效的，對執行 plan 前沒有預期到的事件做應對，相較於 case-based reasoning，比較不會有反應遲鈍或過度反應的狀況發生。然而缺點是，剛開始開發出來的系統，都只能藉由人工手動調整的方式，來更新系統中能夠辨識或反應的狀況(Weber, Mateas et al. 2010)。

為了改進這個缺點，Jaidee、Munoz-Avila 和 Aha 在 2011 年，將 case-based reasoning 和 reinforcement learning 整合進系統中，大大降低了人工調整的需求，讓系統能夠在一場場遊戲中自動學習。不過，他們同時也發現到，一旦將這套系統應用到其他較簡單的場域中，實際效果將不如手動調整的 GDA 系統(Jaidee, Muñoz-Avila et al. 2011)。

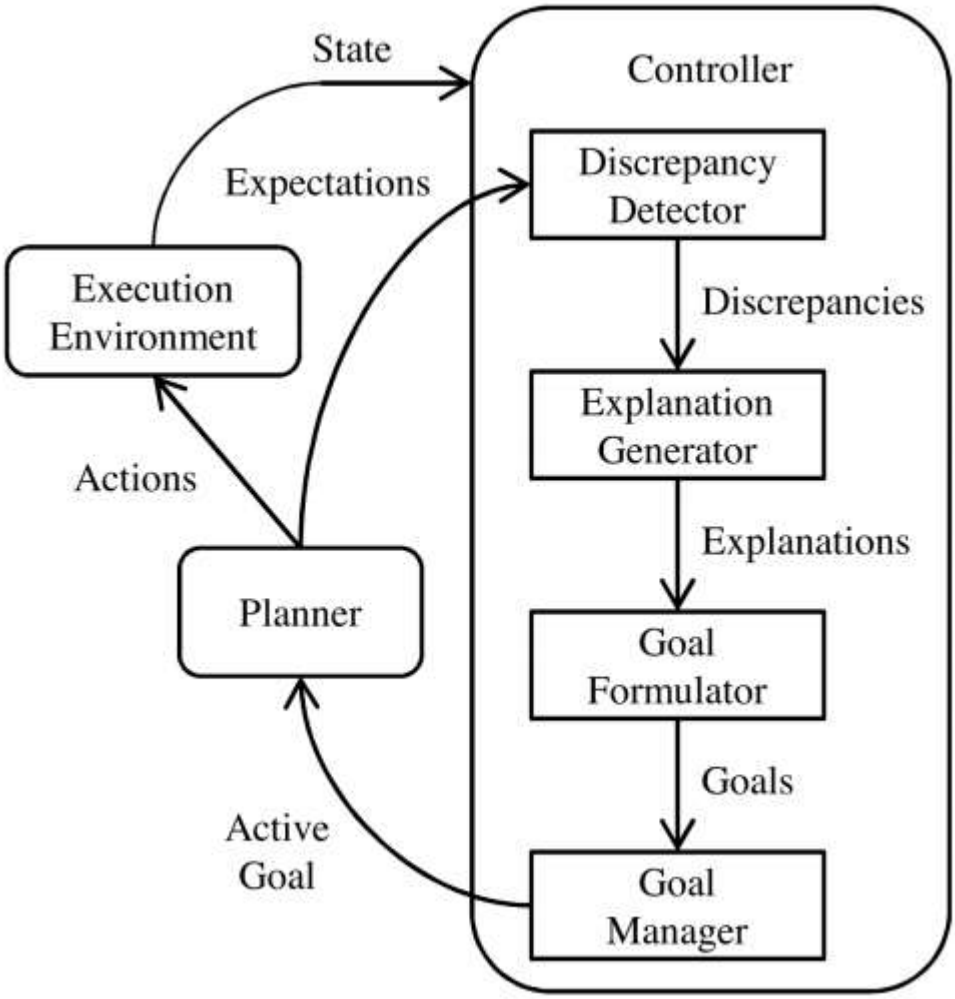


圖 6：GDA 的概念模型(Weber, Mateas et al. 2012)。

(5) State Space Planning

這個方法取徑於比較經典的人工智慧方法，諸如 planning 及 scheduling 等等，先定義好起始 state 和目標 state，接著使用一些 heuristic 找到一連串動作（action），來完成從起始到目標的過程。為了用在即時戰略遊戲上，這套方法必須加入一些新的概念，像是平行動作（parallel actions）、integer-valued state variables 和緊湊的時間限制(Robertson and Watson 2014)。

在 Churchill 和 Buro 於 2011 年的研究中，他們將各個目標（goal）定義為，產生一個用最少的時間，可以達成特定種類和數量的建築和單位的 plan。他們大幅簡化遊戲要素，以增進搜尋的效率，例如在對戰前期完全忽略對手存在，以及將資源採集計算直接

用每個工人的採集速率總和，等等。後來，他們選擇 branch-and-bound depth-first search 來處理 state space 的搜尋，將「資源採集足夠所需時間」和「總共建造時間」取較長者為下界（lower bound），以及一個隨機選擇而達成目標的時間為上界（upper bound）。這個方法使用了大量的專家知識（expert knowledge），但似乎已經是一個能很有效率，在 StarCraft 這個遊戲的前中期，找出接近最佳解的一套方法了(Churchill and Buro 2011)。

(6) Evolutionary Algorithms

演化計算是一種，透過模仿自然演化中的交配和突變等概念，讓人工智慧在不需要太多專家知識（expert-knowledge）的情況下，找尋到一些可用的解。目前雖然沒有研究在 *StarCraft* 上嘗試這個方法，但網路上曾經出現過一套，能夠替 *StarCraft 2*[10]計算出建築物的建造順序的演化計算系統。只要輸入想要完成的單位、建築和科技升級作為目標，系統就能計算出能夠花費最少時間或資源的建造順序(Robertson and Watson 2014)。可惜的是，這套系統目前已經消失在網路上，最後被討論的時間停留在 2010 年左右。

(7) Cognitive Architectures

Cognitive architectures 所採用的概念是，以模仿一個假想的人類心智模型（mental model），包含空間識別系統（spatial visual system）和工作記憶（working memory）等等，建構出一套能夠用類似模式去認知環境，並推論（reason）出後續策略的系統。

在 2012 年，這個方法由 Turner 應用在 *StarCraft* 的 AI 上，使用 Soar 架構來處理部份 strategic decision-making 的問題，例如尋路（path-finding）和記住每個 state 的相關資訊。但它只能夠使用軍營（barracks）和陸戰隊（marine）這些很基礎的建築和單位來作戰，建造建築時的位置擺放也是以事前寫死（hard-coded）的規則決定，沒辦法完全處理整個遊戲中複雜的可能性。[11]

在更早之前其實也有其他人用過這個方法，只不過是應用在 ORTS 上。一樣是採用 Soar[12]這套架構，他們額外加入了一些方法來簡化這個遊戲，例如它一次只能處理一個目標，並且會將一整群鄰近的單位（unit）視為一個單元（entity），這讓整個場面的變數降低，對系統來說較能掌握情況。然而，ORTS 相對於 *StarCraft* 來說，依然還是個太單純的 RTS 遊戲(Wintermute, Xu et al. 2007)。

[10] *StarCraft 2*: http://tw.blizzard.com/zh-tw/games/sc2/
[11] Soar-SC: https://github.com/bluechill/Soar-SC
[12] Soar: http://soar.eecs.umich.edu/

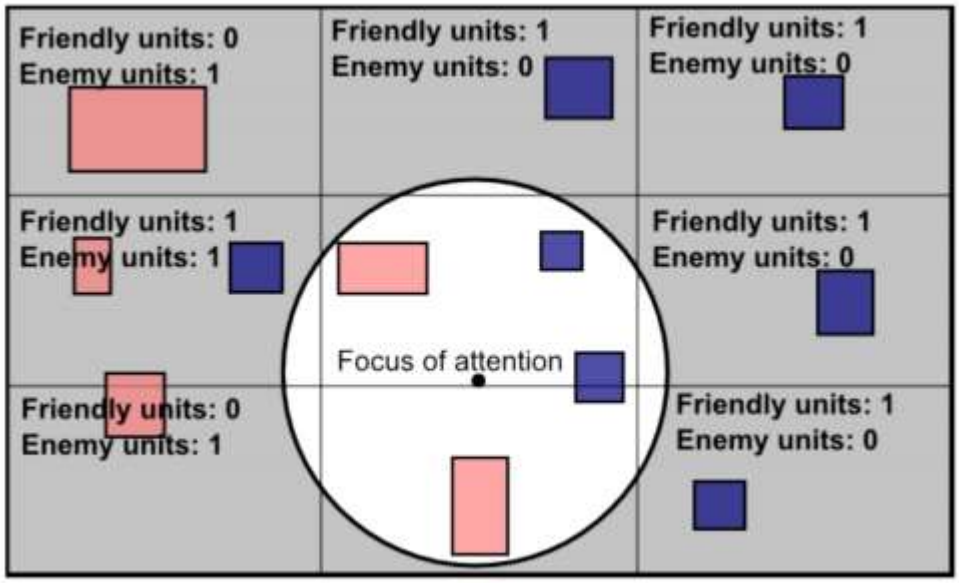


圖 7：能動者（agent）的注意焦點示意圖(Wintermute, Xu et al. 2007)。

(8) Spatial Reasoning

針對 strategic decision-making，還有很重要的一個問題就是地形辨識。玩家為了要分析地圖上各處的戰略意義，在認知上往往會把地圖約略分為幾個區塊（region），以及它們之間的連接處，也就是會因為地形限制而形成無法一次讓大量單位同時通過的窄口（choke point）。Perkins 在 2010 年實作了一套名為 *Broodwar Terrain Analyzer*[13]的函式庫，主要以 Voronoi diagram 的方式來分析地圖，辨識出可步行區域，再將地圖分為數個區塊和窄口。分析一張地圖需要花上大約 43 秒；相較於人眼辨識，它有 0-17%的偽陰性率（false negative rate）以及 4-55%的偽陽性率（false positive rate）(Perkins 2010)。

[13] BWTA: https://code.google.com/archive/p/bwta/

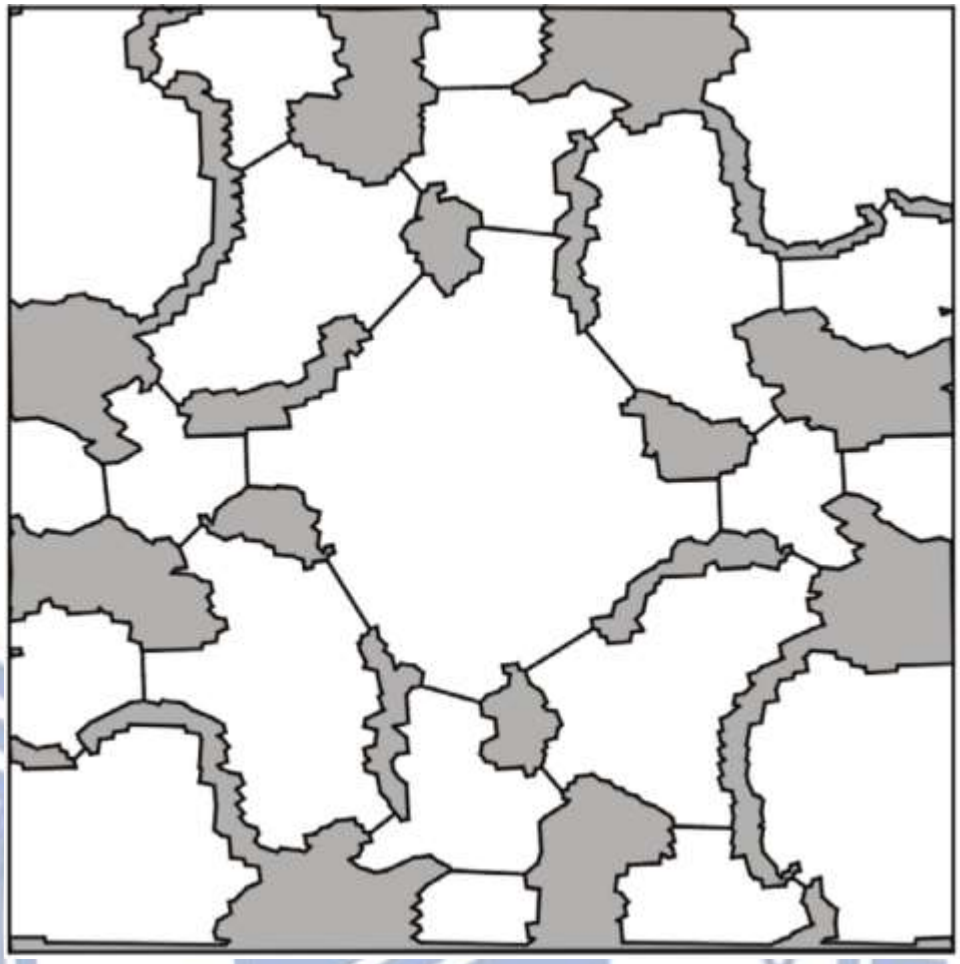


圖 8：分析後的地圖樣貌。灰色區塊是地形阻隔(Perkins 2010)。

能夠分析地形之後，再來就比較有能力預測敵方單位位置了。由於遊戲中有戰霧（fog of war）的設計，在己方單位視野以外的地圖訊息會被隱藏；也就是說，敵方單位在視野以外的動向都是未知的。

Weber、Meteas 和 Jhala 在 2011 年提出了一套粒子模型（particle model），以單位的移動速度，及上一次出現的位置，來預測敵方在離開視野之後的動向。這些預測位置都包含一個信心值（confidence），來表示預測的可信度，而這個值會隨著單位離開視野以後遞減。此外，他們也藉由職業玩家在各張地圖的遊戲紀錄，來建構出專家的移動模型（movement model），讓預測的準確度提高。這個方法，也確實在比賽中提高了對於其他 bot 的勝率(Weber, Mateas et al. 2011)。

另外還有非常多的研究，是採用 potential field 或是 influence map，如此就能夠很輕易的用類似向量場（vector field）的方式塑造出地圖意識（map awareness），讓人工智慧能夠分辨出，哪些地方比較安全，哪些地方又比較可能有隱藏的威脅。(Hagelbäck and Johansson 2008, Kabanza, Bellefeuille et al. 2010) PF 也很適合引入其他策略要素，讓效果更為顯著，例如遠距單位可以「風箏（kiting, hit-and-run）」敵人(Uriarte and Ontanón 2012)、增強軍隊間的協同作戰能力(Baumgarten, Colton et al. 2008)、與 CBP 系統結合(Weber and

Ontanón 2010)等等。將這套方法設計得最有效而精緻的，是一個叫做 *Overmind* 的 *StarCraft* bot，由 Huang 於 2011 年所開發。

2.1.2 Tactical Decision-Making

Tactical decision-making 指的是一些以短程目標（short-term goal）為主的相關決策，例如兩軍短兵相接時的單位指揮，社群中稱為 micro（micromanagement）。在遊戲業界，這類問題通常會以簡單的腳本（script）或有限狀態機（finite-state machine）去解決，就能達到還能接受的效果。但其實在這短時間內所做的決策，有著比輸贏更重要的事：除了要贏，還要以最小成本去贏，盡可能留住戰力。雖然複雜度相對於整個遊戲來說，已經算是比較小，但在各種單位混合作戰的情況下，要找到近似最佳解，依然不是簡單的事(Robertson and Watson 2014)。

(1) Reinforcement Learning

Reinforcement learning 是 machine learning 的一種方式。藉由定義 reward function，agent 就能藉由不斷的嘗試，來調整參數，朝最大化 reward function 趨近。這個方法很適合用在陌生場域中，因為它需要較少的專家知識（expert knowledge），就能讓 agent 自行學會規則與策略。然而，以 RTS 的 tactical decision-making 來說，還是顯得有些複雜，必須配合良好的遊戲抽象化（abstraction），取出真正關鍵的部分，才能讓學習的效果夠好(Robertson and Watson 2014)。

RL 最早於 2011 年被應用在 *StarCraft* 上，但那時還只能針對非常小規模的戰鬥來學習，只有三個單位對上三個單位而已，到了六對六之後就無法戰勝遊戲內建的人工智慧了(Shantia, Begue et al. 2011)。2010 年 Judah 等人曾將 RL 的學習方法加入人類評斷（human critique）應用在 *Wargus* 上，結果顯示，比沒有人類介入的時候效果更好(Judah, Roy et al. 2010)。在更早以前，Marthi 等人認為在 RTS 上，不應該以每個單位來產生行為（action），而是將他們分群之後，每一群做一個行為，如此能夠大大降低複雜度；這影響了之後許多 RTS 相關的研究，也採用了這個概念(Marthi, Russell et al. 2005)。

(2) Game Tree Search

因為 *StarCraft* 本身的封閉特性，我們無法獲知整個遊戲內部的規則與參數，造成使用搜尋相關的方法時，這些產生出來的 state 其實是不完全準確的，而這個誤差會隨著搜尋深度持續累積，讓相關的研究發展遭遇一些瓶頸。另外，為了搜尋的效能，某些誤差的來源幾乎是必須要承擔的，例如單位間的碰撞偵測（collision detection），以及單位移動時的轉向、加速與減速。如何處理這些誤差，便成為了搜尋中重要的問題(Churchill, Saffidine et al. 2012)。

2007 年，Sailer、Buro 和 Lanctot 試圖由賽局理論（game theory）的觀點去切入問題。他們在一個簡化的 RTS 環境中，藉著設定一些已知的策略和簡單的評估函式

（evaluation function），來搜尋遊戲的奈許均衡點（Nash equilibrium），利用這個方式來略過大部分的 branch 和沒有事件發生的 frame，達到大量搜尋的效果。此外，他們也將單位分為群體，針對群體來產生行為（action），而不是每個單位都各自產生。在這個簡化的 RTS 中，他們的實驗證實了這個方法，比用腳本（script）寫出來的 AI，或是 min-max、max-min 的 AI，都來得要好(Sailer, Buro et al. 2007)。

後來在 2012 年，Churchill、Saffidine 和 Buro 撰寫了一套叫做 *SparCraft*[14]的模擬器（simulator），它實作了大部分規則的近似版本，忽略掉單位碰撞等等限制，讓他們可以在遊戲外做模擬或搜尋。它同時能夠跳過那些無關緊要的 state，只搜尋對戰局有意義的部分，例如某個單位做完了一個動作（action），或是有單位進入另一個單位的射程。他們在這套模擬器上設計了一個特殊的 alpha-beta search，並拿來與腳本（script）的 AI 對戰，結果發現，在隨機產生但平衡的戰局下，勝率可以達到 92%。然而，在模擬器上的成功，並無法完全反映在原本的遊戲中。在模擬器上的結果顯示應該能夠 100%擊敗內建的 AI，但實際用在遊戲中，只有 84%的勝率(Churchill, Saffidine et al. 2012)。

(3) Monte-Carlo Tree Search

由於 MCTS 與傳統的遊戲樹搜尋（Game Tree Search）有著很大的不同，所以獨立一個小節出來探討。

MCTS 的主要精神是，既然不可能搜遍所有可能性，那就用統計的方式來推測出最好的下一步。它會以近乎隨機的方式，來將某個選擇模擬到遊戲結束，再將結果紀錄下來，如此往復採樣，就能比較出各個策略的價值。之所以適合用在 RTS 上，是因為它能夠處理巨大的狀態空間（state space），應付得了許多不確定性和對手策略，以及能在時限之內盡可能優化決策，而這對即時性（real-time）的遊戲來說尤其關鍵(Robertson and Watson 2014)。雖然這個方式在許多場域中都促成了突破，例如圍棋，但到目前為止，MCTS 都還沒真正被用在 *StarCraft* 上，只在一些相對簡化的 RTS 中測試過，例如 *Wargus*(Balla and Fern 2009)。

Chung、Buro 和 Schaeffer 在 2005 年設計了一款搶旗遊戲（capture-the-flag），來測試 MCTS 在 RTS 中的效果。結果大致來說是肯定的，但到底能夠達到多高的強度卻很難驗證，因為是自己設計的遊戲，並沒有一個大家普遍認可的腳本（script）AI 來當作對手；此外，他們的方法非常依賴 heuristic evaluation function 來評估中介狀態（intermediate states）以做出適當決策(Chung, Buro et al. 2005)。

後來在 2009 年，Balla 和 Fern 將 UCT（Upper Confidence Bounds applied to Trees）這個比較晚近的 MCTS 技術應用在 *Wargus* 的小規模戰鬥上。他們所採用的方法，最大的優點就是，它並不需要一個 heuristic evaluation function 來幫助決策，而是完全以隨機的方式來模擬完一個選擇。實驗設定為雙方有等量的步兵（footman）散布在空曠的地圖

[14] SparCraft: https://github.com/davechurchill/ualbertabot/tree/master/SparCraft

上，結果是這個 AI 非常顯著的擊敗了所有腳本（script）AI，以及人類玩家(Balla and Fern 2009)。

(4) Case-Based Reasoning

CBR 是一個藉由過去的類似已知狀況，來解決眼前問題的方法。雖然它時常被用在 RTS 的 strategic decision-making 上，但在 2008 年的時候，也曾被應用到 *Warcraft III* 的 tactical decision-making 上。Szczepanski 和 Aamodt 替每一個 case 都加入了條件限制，來確保每次選擇到的 case 都是能夠馬上執行的；此外，他們也設計了 reactionary case 來精簡策略生成的過程，一旦某些情況發生，就直接採用這類 case 的處理方式。實驗結果顯示，這個 AI 確實能夠在小規模的作戰中擊敗內建 AI(Szczepański and Aamodt 2008)。

(5) Neuroevolution

Neuroevolution 是演化演算法的一種，以建構出神經網路（neural network）來模仿人類神經元的運作方式。Gabriel、Negru 和 Zaharie 透過 rtNEAT 這套 neuroevolution 方法，來訓練 *StarCraft* 中的個別單位控制。神經網路的輸入來自環境，例如障礙物和其他單位，以及他們自行定義的一些抽象化的遊戲資訊，而相對應的輸出選擇，則有攻擊和撤退等等。訓練過程中，他們使用了自行設計的 fitness function 來評估每個 agent，比較差的會被表現好的取代掉。實驗設置方面，則有 12 對 12 的小規模戰鬥，分別嘗試了純近戰（melee）單位，以及純遠距（ranged）單位，結果都能夠戰勝內建 AI。但截至目前為止還不確定的是，在不同類型單位的混合編隊下，效果是不是還能一樣好(Gabriel, Negru et al. 2012)。

三、 研究方法

3.1 研究工具

本研究採用 *BWAPI (Brood War Application Programming Interface)*來實作研究中的所有內容。*BWAPI* 為一款由第三方所開發的 C++ framework，免費而且開放原碼，讓研究者能藉由程式來即時取得 *StarCraft* 中的各項資訊，例如建築與單位的分布、單位生命值及各項屬性等等，並且直接對遊戲下達控制指令，而不只是仿造鍵盤輸入。

3.2 遊戲抽象化

為了方便程式做計算及模擬遊戲規則，設定好一套抽象化的架構是非常重要的。雖然遊戲強調即時性（real-time），但實際上遊戲在電腦上運作，本身就無法避免的有離散化的特性。遊戲內部的時間就等同於 frame time，根據遊戲內部運作的狀況，每隔幾個 frame 就會去輪流接收各玩家的輸入（input），而這就是我們的程式能夠做判斷及下達指令的時間。

3.2.1 Unit

- ID：此單位的編號。
- isAlly：是否為己方單位。
- Unit Type：單位的類型，例如 marine（陸戰隊）、zealot（狂戰士）。
- Position：單位所在的像素位置，大略以模型中心為準。
- Hit Points：單位的生命值；在遊戲中，歸零即會造成單位死亡。
- Shields：單位的護甲值；護甲被損傷到歸零，才會傷害到生命值。
- tAttack：下一次能夠攻擊的 time frame。
- tMove：下一次能夠移動的 time frame。

只要有單位類型，就能透過 *BWAPI* 取得許多單位的屬性，像是移動速度、武器類型、武器冷卻時間和最大生命值等等。

tAttack 及 tMove 的設計則是方便推算下一個有事件發生的 frame time，意即「場上有任何一個單位可以做出新動作的時間點」。這也讓程式能夠很容易產生出接下來合法的動作（legal actions），因為對一個單位來說，當目前的 time frame 大於等於 tAttack 或

tMove，就分別代表了它現在可以攻擊或是移動(Churchill, Saffidine et al. 2012)。每次攻擊時，就以當下時刻為準，將 tAttack 加上武器冷卻時間，tMove 加上攻擊動畫長度，否則在這之前下達其他指令，都會讓攻擊中斷，傷害不被計算(Churchill and Buro 2012)；移動時，就將兩個數值分別加上預設的一小段時間，在本研究裡是固定為 8 frames，也就是 0.3 秒左右(遊戲為 24 frames/sec)；停止指令則是根據各個情況，將 tAttack 和 tMove 都設到停止結束的那個時間點。

3.2.2 Action

- Unit ID：執行此動作的單位編號。
- Action Type：動作的類型。
- Target ID：如果動作牽涉到目標，此為目標單位的編號。
- End Time Frame：如果動作牽涉到結束時間，此為結束的 time frame。

本次研究中包含的動作類型，包含普通攻擊（Attack）、移動（North、East、West、South）以及停止（Stop），還不包括一些特殊的技能（skills）及魔法（magic）。對於移動，程式只計算往四個方向移動固定時間（8 frames，相當於 0.3 秒，因為遊戲本身固定 24 frames/sec）的可能性，而遊戲裡提供的操作彈性事實上應該還遠超過八個方向，會如此簡化主要是為了減少 branching factor。

除了前兩項必填之外，後兩項都是根據動作類型來決定是否選填。例如：

- 〈0, Attack, 2, -1〉= 編號 0 的單位攻擊編號 2
- 〈0, West, null, -1〉= 編號 0 的單位向西一步（長度固定 8 frames）
- 〈0, Stop, null, 50〉= 編號 0 的單位停止在原地直到遊戲時間達 50 frames

在程式中所搜尋的樹，其中的節點就是儲存著一組組的 Actions，來代表從親節點（parent node）執行這組 Actions 會產生這個節點。因此，每個節點之後的可能性，就是由每個單位所能夠做的動作，相互排列組合而成。

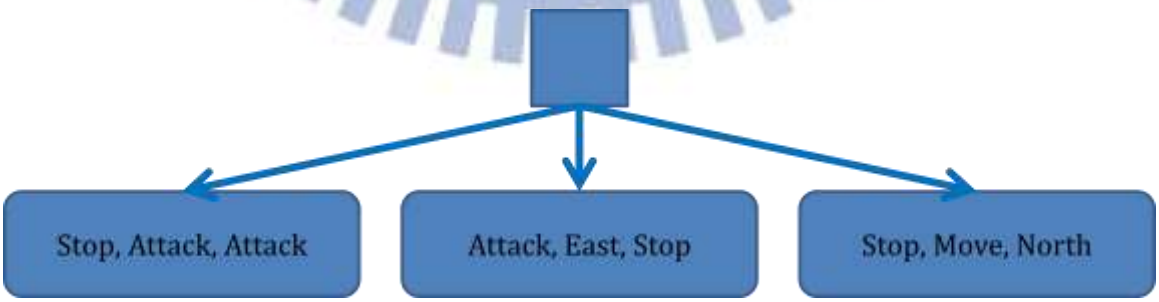


圖 9：節點與 Actions。以三個單位來說，一個節點底下的可能性，就是由這三個單位各自能做的動作，互相排列組合而成。上圖僅為舉例，實際上還能產生更多的子節點。

3.2.3 State

- Current Frame：此 state 所在的 time frame。
- All Units：儲存場上的所有單位，包含敵我雙方。

每當要開始搜尋之前，都會先從真實的戰場上，將所有單位資訊更新一份過來。有了抽象化的複本後，就能每次複製一份，從 root 開始，選擇節點或者產生節點，根據節點內儲存的 Actions 模擬出行動之後的 State，再重複同樣過程到子節點。

3.2.4 模擬限制

由於遊戲本身是封閉的商業遊戲，完整的規則模擬是無法做到的；同時，為求做樹搜尋（tree search）時的效率，一些適當的近似也是必要的，以 MCTS 的統計性質來說，預想上並不會造成太大的影響。

- 生命值（hit points）和護甲值（shields）不會自然恢復。
- 子彈、飛彈等等投射物（projectiles）沒有飛行時間，直接計算傷害。
- 不處理單位碰撞；這會是拖慢效能的最大主因，基本的單位碰撞演算法都需要 $O(n^2)$的時間複雜度，同時我們無法得知精確的模型形狀與大小。
- 不處理單位加速、減速和轉向等等動作時間，一律以最大移動速度計算。
- 無戰霧（fog of war）設置；實際作法是在製作地圖時就揭露所有視野。

如果不處理單位碰撞，對遠距（ranged）單位來說應該不會有太大差別，因為它可以在碰撞到敵人以前就發動攻擊；但對近戰（melee）單位這種會產生大量碰撞的來說，我們需要藉由實驗來確認，這樣的模擬是否依然有效。

3.3 Monte Carlo Tree Search

3.3.1 UCTCD

UCT（Upper Confidence Bound 1 applied to trees）為 MCTS 的其中一種形式。探勘（exploitation）與探索（exploration）的平衡一直是搜尋的重大議題，在 MCTS 中尤其重視。探勘（exploitation）代表了多經由眼前已經顯示是好的解，繼續往它的更深處搜尋；探索（exploration）則是指嘗試那些還未經搜尋或是極少搜尋的節點，或許往別的方向搜尋能夠有機會找到更佳的解。而 UCT 試圖利用一道公式，用於選擇下個欲展開的節點，來平衡上述兩者：

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}} \qquad (1)$$

- w_i：經由節點 i 的累積勝場數。
- n_i：經由節點 i 的累積模擬次數。
- c：exploration parameter，用來控制前後項的權重比例的常數。
- t：經由親節點的總模擬次數，也就是其所有子節點模擬次數的總和。

UCT 會根據這個公式，挑選整體數值最大的節點，來做展開和模擬。前項是這個節點 i 的勝率，代表它的價值，也就決定了 exploitation 的傾向；後項則會因為節點 i 的累積模擬次數被放在分母，對模擬次數較少的節點來說，後項會越大，也就決定了 exploration 的傾向。常數 c 則需要在不同的場域中做調整，可以由實驗求得。

傳統的 UCT 都是用於回合制（turn-based）的遊戲，但是 *StarCraft* 是一個即時制（real-time）的遊戲；也就是說，不會像下棋一樣，一個玩家動了一步之後，還要等對方動完才能走下一步，而是盡可能的在同樣時間內，下達越多對戰局有正面影響的指令越好。為了將樹搜尋應用在即時制的遊戲中，我們必須引進時間的因素到每個動作上。

UCTCD（UCT Considering Duration）讓每個動作都有它的開始與結束時間，以及允許同時發生的動作。例如，攻擊動作會由一連串的動畫所構成，所以這段動畫一直到傷害被計算，就是它的動作時間；移動相關的動作都是預設 8 個 frame（0.3 秒）；等待動作則可以根據需要設定結束期限(Churchill and Buro 2013)。

每個節點除了儲存動作以外，也會加上一個分類來辨別是否和對手的動作同時發生。SOLO 表示這個時間點只有一方能夠動作，FIRST 及 SECOND 則分別代表一組同時動作的我方和敵方。被標示為 FIRST 的動作，除非已經是末端的 leaf，否則不會在拜訪時馬上更新，而是會等到拜訪被標示為 SECOND 子節點時，才會一起更新，否則會產生延遲動作效應（delayed action effect）：明明是同時發生，先手卻占有極大的優勢，例如擊殺了對方一個單位，對方就少一個單位能做攻擊，這是不符合遊戲真實狀況的(Churchill and Buro 2013)。

UCTCD 不會直接將一個節點模擬至遊戲結束，而是會以遞迴（recursive）的方式反覆從 root 往下挑選節點，並且在抵達 leaf 時產生新的節點。由於整個狀態空間（state space）真的太過龐大，我們可以透過使用一些評估函式（evaluation function）來衡量那些非終端的狀態（non-terminal state）。

關於評估函式，一個最直觀而且被大量使用的方式就是，雙方單位總生命值（hit points）的差。然後我們要引進一個稱作「每幀傷害（Damage Per Frame, dpf）」的指標，假設這個單位每隔一段完整的武器冷卻時間，都會攻擊一次的話，那它的傷害除以它的

武器冷卻時間，即為它的 dpf。定義了 dpf 以後，將它乘上該單位的當下生命值，我們稱為「生涯造成傷害（Life-Time Damage, LTD）」，提供除了生命值以外的資訊，希望能更有效的去評估單位的價值。再來，如果將生命值開根號取常態分佈，又能得到新的函式 LTD2(Churchill, Saffidine et al. 2012)。

$$\mathrm{HPD}(\mathrm{state}) = \sum_{u \in U_{ally}} \mathrm{hp}(u) - \sum_{u \in U_{enemy}} \mathrm{hp}(u) \tag{2}$$

$$\mathrm{dpf}(\mathrm{unit}) = \frac{\mathrm{damage}(\mathrm{unit})}{\mathrm{cooldown}(\mathrm{unit})} \tag{3}$$

$$\mathrm{LTD}(\mathrm{state}) = \sum_{u \in U_{ally}} \mathrm{hp}(u) \cdot \mathrm{dpf}(u) - \sum_{u \in U_{enemy}} \mathrm{hp}(u) \cdot \mathrm{dpf}(u) \tag{4}$$

$$\mathrm{LTD2}(\mathrm{state}) = \sum_{u \in U_{ally}} \sqrt{\mathrm{hp}(u)} \cdot \mathrm{dpf}(u) - \sum_{u \in U_{enemy}} \sqrt{\mathrm{hp}(u)} \cdot \mathrm{dpf}(u) \tag{5}$$

最後，在限定的時間用完時，演算法會回傳第一層的節點中，被拜訪最多次的那組動作，交由負責下達指令的模組去執行。

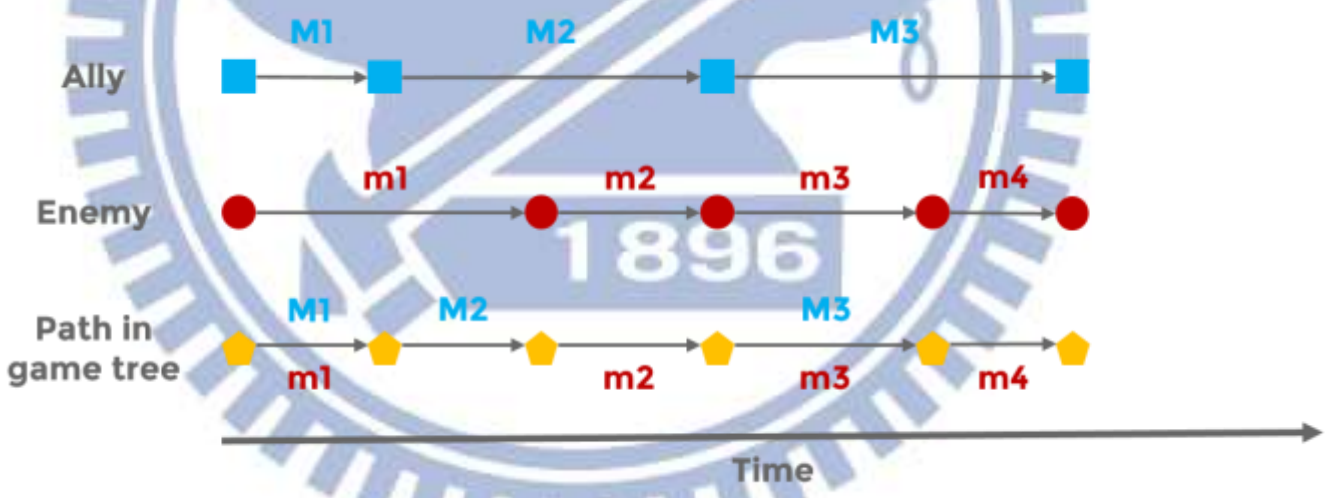


圖 10：Durative actions 的範例示意圖。由圖中可以看出，雙方做出新動作的時間點不盡相同，卻依然能夠藉由同時更新雙方動作，建構出遊戲隨著時間進行的架構，讓我們能夠應用樹搜尋相關的演算法。

3.3.2 Actions Generation

由於搜尋空間實在太龐大，尤其是在單位很多的情況下更是呈指數成長，如果只是完全隨機的產生新動作（actions），效果一定不會太好。這時候我們就需要借助一些已經被證實有效的腳本（script）來幫助我們產生動作。

本次實驗中使用了兩個腳本，NOK-AV（No-OverKill-Attack-Value）以及 Kiting-AV（Kiting-Attack-Value）。所謂 attack value，是為了挑選出當下最有價值的攻擊目標，而衡量方式就是使用前述的 dpf 除以它的生命值。如此一來，dpf 越高、生命值越少的單位，就會優先被選上(Churchill, Saffidine et al. 2012)。

$$\text{AttackValue}(\text{unit}) = \frac{\text{dpf}(\text{unit})}{\text{hp}(\text{unit})} \tag{6}$$

NOK-AV 的策略為，如果攻擊範圍內有敵人，而且可以攻擊，就攻擊範圍內 attack value 最高的敵人；如果範圍內有敵人，武器卻仍然在冷卻中，無法攻擊，就待在原地直到冷卻結束；如果範圍內沒有敵人，則朝最接近的敵人靠近一步。此外，這套腳本會記錄下每個敵人已經被分配到的傷害，如果已經被分配到足以致命的傷害，還可以攻擊的友方單位就會選擇下個順位的敵人做攻擊，不會浪費傷害。Kiting-AV 的策略和 NOK-AV 非常接近，差別只在於，當範圍內有敵人，武器卻仍在冷卻，無法攻擊，它就會選擇遠離最近的敵方單位一步(Churchill, Saffidine et al. 2012)。

每次對節點產生新的子節點及動作時，最開頭的兩個動作將會依序為 NOK-AV 和 Kiting-AV。而第三個子節點，會從 NOK-AV 中隨機挑選一個非選擇攻擊的單位，隨機讓它選擇一個和原本所選不同的方向走一步。之後的每個子節點，都會以前一個子節點作為基底，重複同樣的隨機產生程序。這樣一來，我們能確保順序比較前面的動作，都是能夠適用於大多數狀況的；越後面產生的，就成了這個演算法的彈性，讓它能夠藉由 UCT 適時的選擇非腳本產生的動作(Churchill, Saffidine et al. 2012)。

以上所談到的都是對於友方單位的動作產生。在模擬的過程中，所有敵方單位所產生的動作，都只會依照 NOK-AV 一種來產生，讓這個部分降階成確定性（deterministic）的問題，也能減少 branching factor，大幅增加搜尋效能。

圖 11：為新的子節點產生動作的規則示意圖。

四、 實驗分析

4.1 實驗設置

4.1.1 硬體環境

所有實驗都是在同一台機器上以單一執行緒跑出來的，它的硬體規格及環境為：Intel(R) Core(TM) i5-4460 CPU @ 3.2 GHz、DDR3 RAM 8 GB、Windows 7 64-bit。程式以 C++ 撰寫，由 Visual Studio 2013 編譯而成。

4.1.2 地圖設定

所有實驗地圖皆為 Scmdraft2[15]這款由第三方開發的地圖編輯器完成。每張地圖都盡可能做到對稱，雙方分別會擁有等量的同種類單位組合。每種組合都有不同單位數量的版本，依序為 4、8、16、32、50 個單位，4 個單位是模擬極小規模的零星戰鬥，而 50 個單位則已經是這個遊戲所可能發生的，最大規模的會戰了。如果沒有另外強調，實驗的對手皆為 *StarCraft* 內建的 AI 玩家，其中沒有難度調整的選項。

單位組合分成三種，分別為 marines、zerglings 和 zealots + dragoons。marines 中文翻譯為陸戰隊，屬於低生命值的輕型遠距單位；zerglings 可以說是某種小型異形，屬於低生命值的輕型近戰單位；最後一個是多種類單位的組合，zealots 為生命值較高的中型近戰單位，dragoons 則是生命值也偏高的中型遠距單位，體型也是實驗中最大的。由於最後一個組合有兩種單位，在單位總數不變的情況下，會均分數量給兩種單位。

勝利條件為擊殺對方所有單位即為勝利，同時死亡視為平手，時間結束也算是平手。時間限制皆為一場一分鐘，除了 zealots + dragoons 這個實驗組合的 16、32、50 單位時限比較長，一場一分半，因為單位生命值比較高，會需要比較多時間。

每張地圖都設定為完全視野，也就是沒有戰霧（fog of war）來隱藏地圖資訊，雙方都能完全清楚整張地圖上發生的事情。雙方能操控的單位除了初始設定的以外，不再有任何其他來源能夠增加單位數量或種類。地圖上也不會有任何障礙物及地形變化。遊戲開始時，不會有任何單位已經在任一敵方單位的攻擊範圍內，否則這個問題將會降階成只在處理「攻擊目標選取」或是「傷害分配」。

[15] Scmdraft2: http://www.stormcoast-fortress.net/cntt/software/scmdraft/

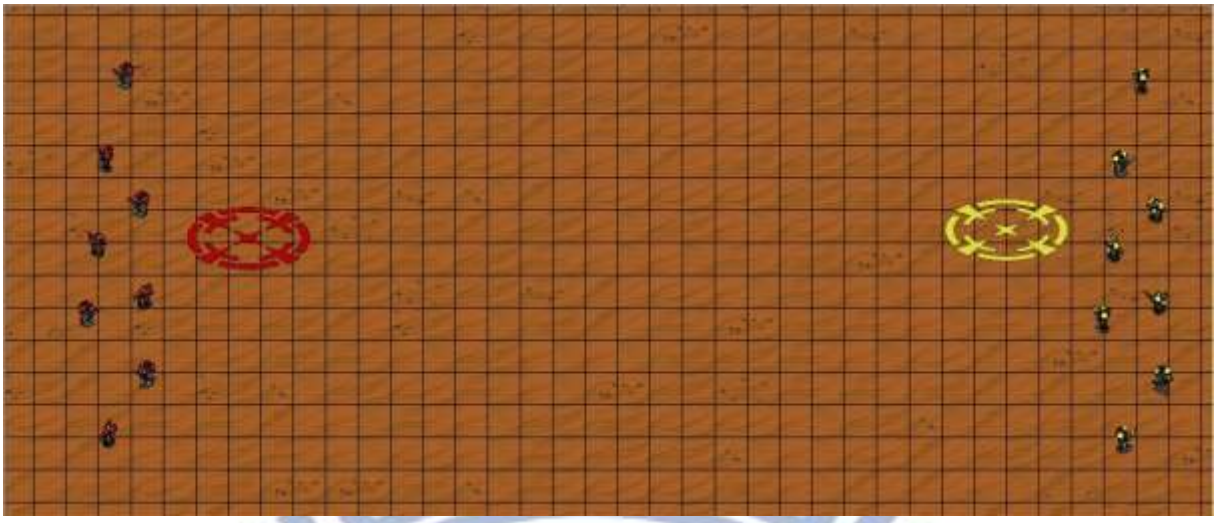


圖 12：8 marines 的自訂地圖。

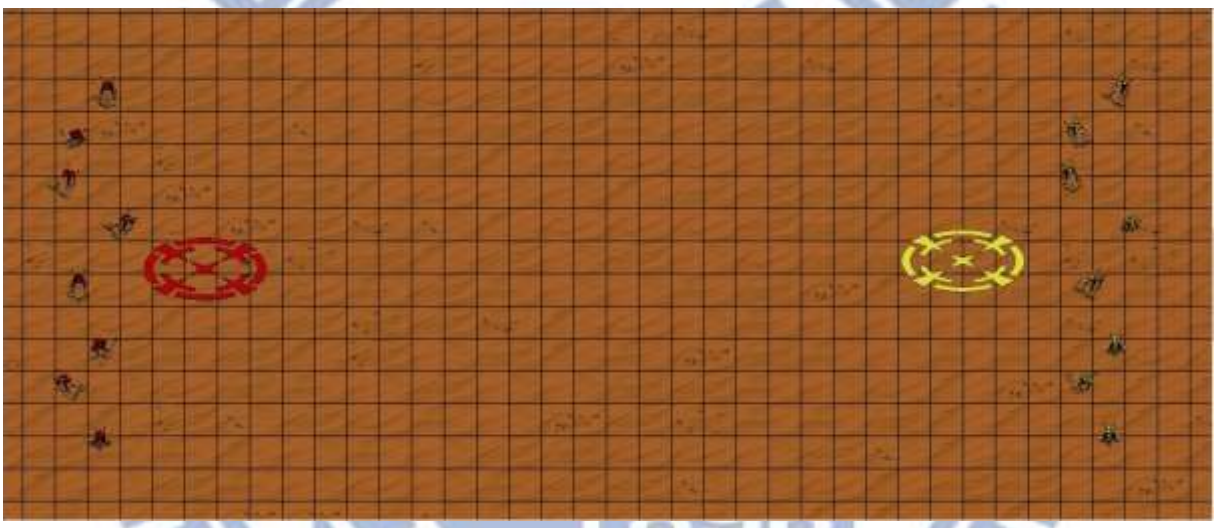


圖 13：8 zerglings 的自訂地圖。

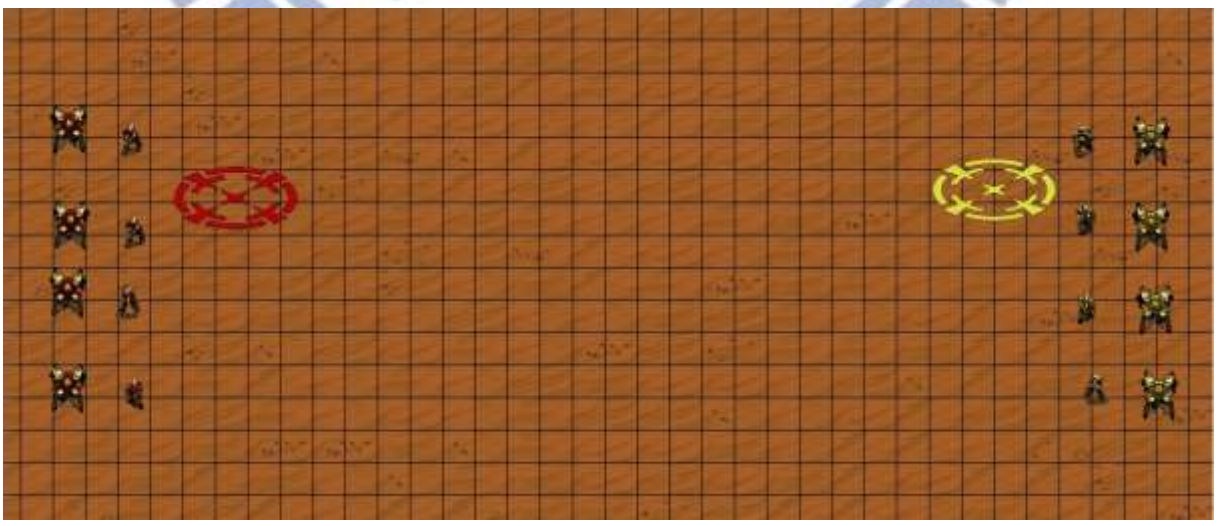


圖 14：8 zealots + dragoons 的自訂地圖。

4.1.3 搜尋時限

透過 *BWAPI*，AI 大約可以在每 3 個 frames 左右獲得一次下達新指令的機會。為了不拖慢遊戲速度，搜尋的時間限制是必要的。在本研究中，我們參考了每年在這個遊戲上舉辦的學生 AI 競賽 *SSCAIT*（Student StarCraft AI Tournament）[16]的規定，超過 360 個 frames 使用了大於 55 毫秒即為失敗，在考慮到其他運算所需的時間，保險的設定為 50 毫秒，超過 50 毫秒 UCTCD 便會停止搜尋，回傳到目前為止拜訪過最多次的動作組（Actions）。

4.2 參數調校

4.2.1 Exploration Constant

Exploration constant 這個固定參數，決定了 exploitation 和 exploration 之間的平衡，這個值越大，UCT 的過程中就越容易去選擇模擬次數較少的節點。在主要實驗開始之前，我們必須先藉由事前實驗，給它一個合理的值。其中，為了避免評估函式間的需求不同，各個評估函式我們都一併測試。相關參數設定如下：

- 單位數量：16
- 單位種類：marines（輕型遠距單位）
- 子節點數目：20

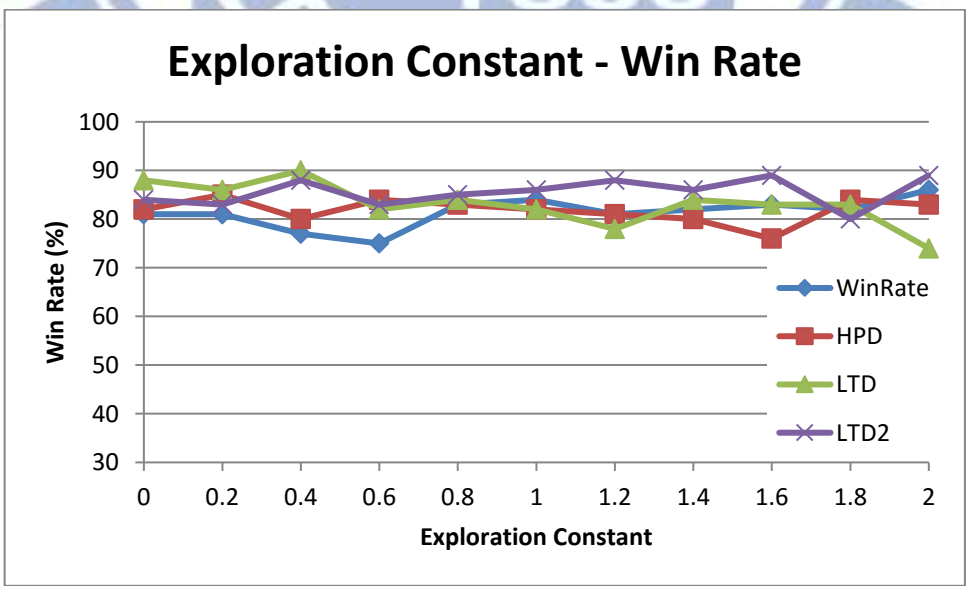


圖 15：各評估函式的 Exploration constant 與勝率之間的關係。

[16] SSCAIT: http://sscaitournament.com/

單位數量之所以取 16，是因為這算是在遊戲中最常碰到的戰鬥規模，屬於中小型會戰；單位種類使用的是輕型遠距單位，因為 UCTCD 的模擬器並沒有考慮單位碰撞，我們還不能肯定在大量單位碰撞產生的情況下，UCTCD 的效果是否能夠一樣好，故先挑選遠距單位做調校；子節點數目則是參考自既有的相關研究結論(Churchill and Buro 2013)，但在下節會為了本實驗再做一次調校。每一個 exploration constant 的值都會實驗 100 場，統計出的勝率如上圖。

從實驗結果來看，exploration constant 的大小並不會帶來決定性的差別，不同的評估函式對此常數的需求也沒有太大差異，大部分的勝率都集中在 80%上下。如果我們再將其取平均（如下圖），依照這個結果，我們設定之後的實驗，都使用 0.2 作為 exploration constant 的值。

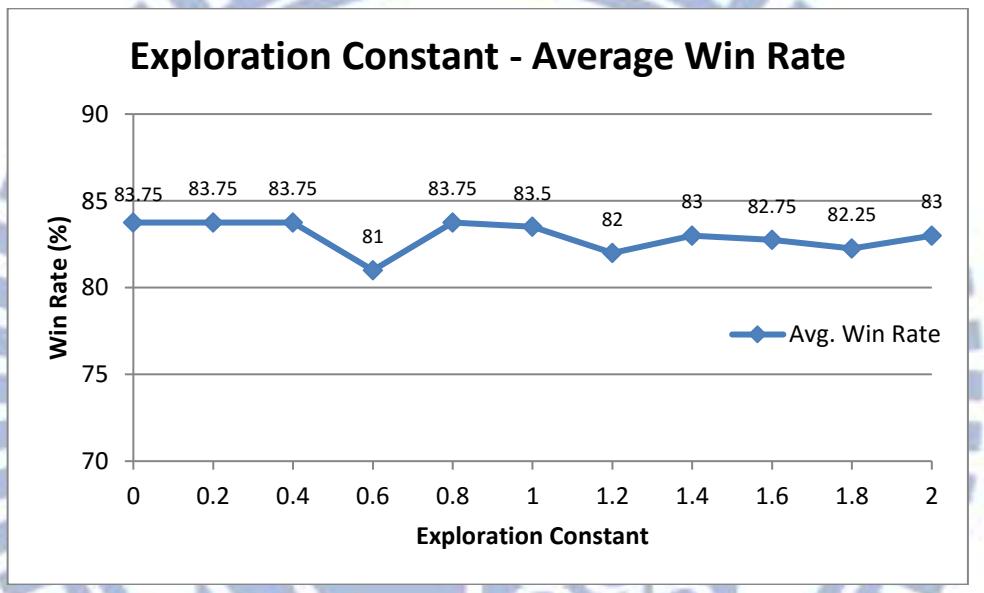


圖 16：Exploration constant 與平均勝率之間的關係。

4.2.2 子節點數目

調整完 exploration constant 之後，我們也需要對子節點數目做類似的測試調校。對搜尋空間這麼大的遊戲來說，在有時間限制下，如果沒有限制每個節點所能產生的子節點上限，UCT 的搜尋範圍可能會幾乎停留在淺層，廣度很廣卻無法深入評估，效率也就很差。相關測試的設定如下：

- 單位數量：16
- 單位種類：marines（輕型遠距單位）
- Exploration constant：0.2
- 評估函式：HPD（雙方總生命值的差值）

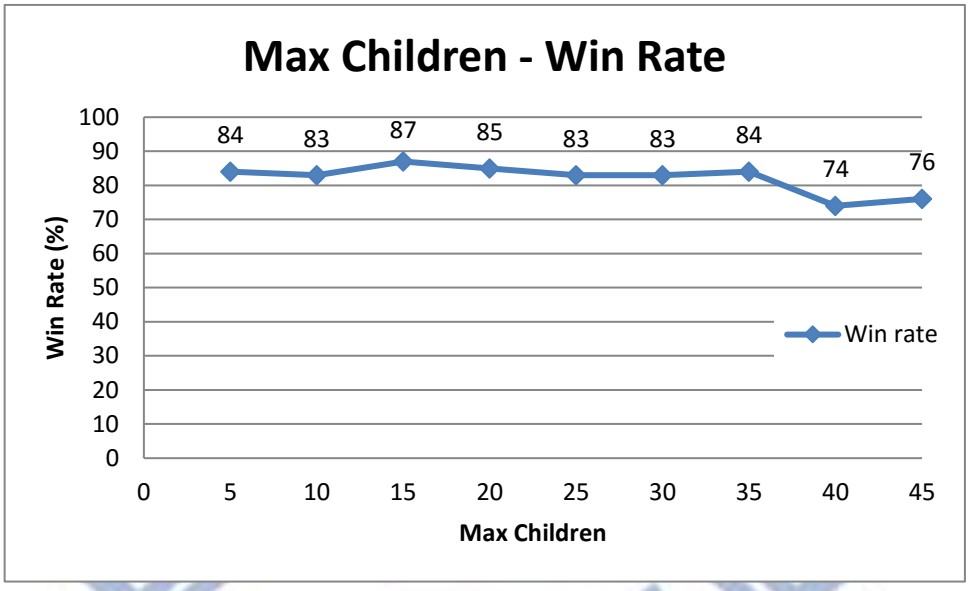


圖 17：最大子節點數目與勝率之間的關係。

由實驗結果可以看出，最大子節點數量到 35 個為止都沒有太明顯的起伏，但超過這個數量，從 40 個開始，就明顯看出勝率因為數量太大而下降。依據這個結果，在之後的實驗裡，最大子節點數量都將設定為 15 個。

4.3 綜合測試

經由前面的參數調校，我們已經可以知道基本的參數該如何設定，而接下來的實驗則是要更進一步的探討，不同的評估函式在各個對戰組合中的表現狀況為何。相關的參數設定為：

- Exploration constant：0.2
- 最大子節點數量：15

針對每個評估函式，我們會測試它們在三種單位組合上的表現，這些組合有 marines（輕型遠距單位）、zerglings（輕型近戰單位）和 zealots + dragoons（中型近戰單位 + 中型遠距單位），而這三種單位組合又分別會對應到五種單位數量 4、8、16、32、50，並且每項測試都會與內建的 AI 對戰 100 個場次來計算勝率。下表為完整實驗結果，而在後續的內容中，我們將分成幾個小節來解析這些結果。

地圖配置		Evaluation Function			
單位種類	單位數量	WinRate	HPD	LTD	LTD2
Marines	4	74	97	38	77
	8	71	85	52	67
	16	84	91	93	86
	32	87	86	90	89
	50	66	70	71	69
Zerglings	4	15	18	15	11
	8	13	13	8	1
	16	10	10	6	7
	32	6	8	3	8
	50	26	34	24	21
Zealots + Dragoons	4	60	68	76	72
	8	56	53	39	53
	16	48	66	64	69
	32	15	12	13	9
	50	2	7	8	5

表格 2：各個 Evaluation Function 在不同對戰組合及數量時的對戰勝率。

4.3.1 單位種類

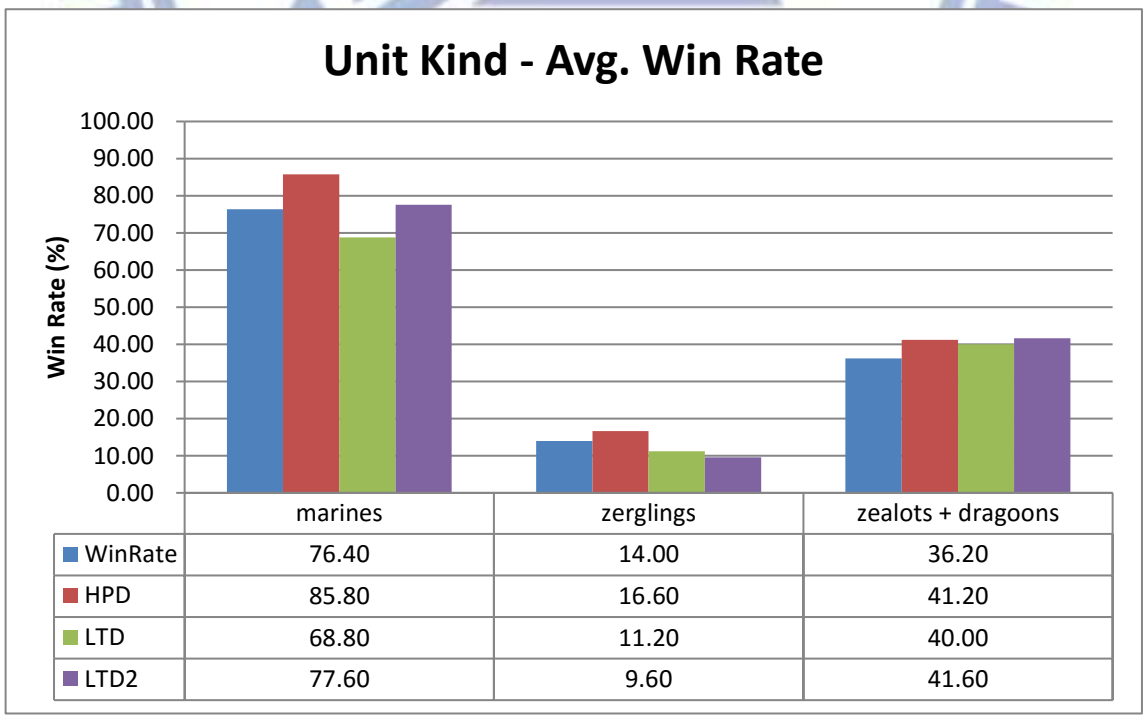


圖 18：各個評估函式在不同單位種類下的平均勝率。

從平均勝率來看，四種評估函式分別在三種對戰組合中的表現都沒有極大差異，而其中以 HPD（單位總生命值的差值）的表現最好。再來，我們如果去比較不同組合間的表現，就會發現，UCTCD 在 marines 中的表現優異，在 zealots + dragoons 稍差，而在 zerglings 則是非常不理想。

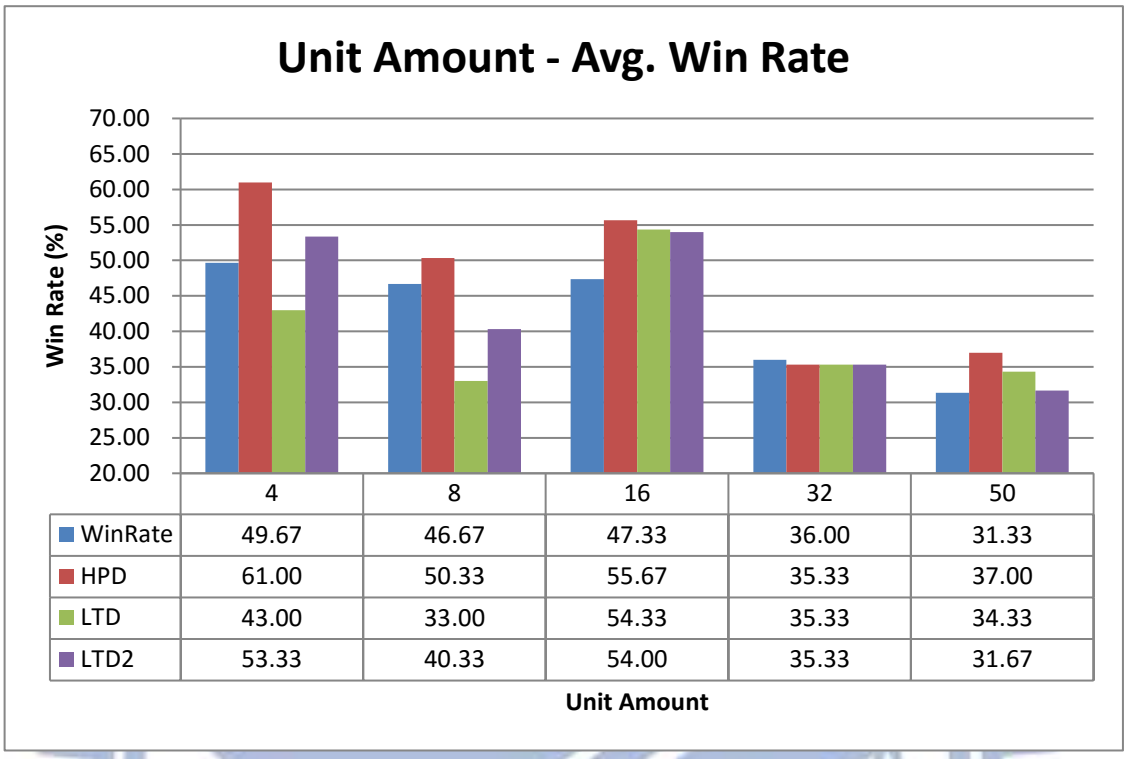
使用 UCTCD，HPD 在 marines 中達到 80%以上的平均勝率，顯示出這個方法對於遠距單位的控制已經超越內建 AI 太多；然而，在近戰單位參與的對戰組合中，對於 zerglings 的操作完全受到內建 AI 的壓制，只有 10%左右的勝率，而在一半為近戰單位的 zealots + dragoons 中，勝率也只剩下大約 40%，平均來說，在這方面的強度是不及內建 AI 的。

可能的原因是，由於 UCTCD 在模擬未來狀態（states）時，為了效能不得不放棄碰撞偵測（collision detection），意即假設所有單位失去自身的模型大小，不占有任何空間，以致單位之間允許重疊，而如此的模擬對於實際的遊戲策略是無法產生效用的。

單位碰撞之所以重要，在於近戰單位需要靠近到敵方單位身邊才能發動攻擊，許多的互相卡位和包圍就在這時發生：該如何在大量單位之間，讓出退路讓友方瀕死單位撤離，健康的單位又馬上補上位子避免敵方追擊；或是，該以如何的陣型包圍敵人，讓己方有最多單位能夠攻擊，同時敵方單位又因為自相推擠無法竄到前線造成傷害。這些策略，都因為模擬器忽略單位碰撞，而無法實現。

圖 19：白色與綠色的 zerglings 在對戰中互相包圍。黃色圈起的白色單位，在走近敵方綠色單位前是無法發動攻擊的。

4.3.2 單位數量



	4	8	16	32	50
WinRate	49.67	46.67	47.33	36.00	31.33
HPD	61.00	50.33	55.67	35.33	37.00
LTD	43.00	33.00	54.33	35.33	34.33
LTD2	53.33	40.33	54.00	35.33	31.67

圖 20：各個評估函式在不同單位數量下的平均勝率。

從圖表上可以看出，無論是哪個評估函式，隨著單位數量增加，平均勝率就隨之降低。即使是表現最好的評估函式 HPD，在單位數量增加到 16 個的情況下依然能有超過 50%的勝率，但接著再增加到 32 個之後，勝率就大幅下降到只剩 35%左右。

可能的因素是，由於 UCT 對於每個 action 的評估方式是藉由大量模擬，在時間限制相同、硬體效能相同的情況下，單位數量就影響到了每次模擬所需的時間，讓整體來說就減少了總模擬次數，模擬次數少了，就會降低 UCT 評估結果的準確性。另一方面，單位數量增加，代表的也是搜尋空間的膨脹。以一個單位可以執行一個 action 來說，50 個單位所排列組合出的可能性，就一定比 4 個單位時來得多上許多。這時，我們所使用的最大子節點數量（15 個），所涵蓋到的可能性就相對變窄了；然而僅僅增加這個數量也無法解決問題，只是把已經有限的模擬次數，分散給更多節點而已。

五、 研究結論

本研究的目的在於，以具有代表性的一款商業遊戲 *StarCraft*，來驗證 Monte Carlo tree search 用於即時戰略遊戲（real-time strategy）中 tactical decision-making 的可行性。MCTS 這個方法本身具有統計採樣的性質，藉由大量模擬眼前的選擇，來評估這些選擇的優劣，所以可望能夠降低對於專家知識（expert knowledge）的依賴，依然達到令人滿意的效果。傳統 MCTS 是設計給回合制（turn-based）遊戲的，為了將它應用到即時制（real-time）遊戲上，我們採用了 UCTCD（Upper Confidence Bound 1 applied to trees Considering Duration），讓每個 action 具有自己的作用時間，並且允許幾個 action 同時作用，依然能建出一顆樹以供搜尋(Churchill and Buro 2013)。相對於 strategic decision-making，tactical decision-making 因為有即時性的限制，又需兼顧到每個決策的執行細節，事實上是較難處理的問題。

其中，我們做了關於評估函式（evaluation function）對於不同單位種類、單位數量之間的實驗。這些評估函式分別為 HPD、LTD 和 LTD2(Churchill, Saffidine et al. 2012)，而單位種類則有 marines（輕型遠距單位）、zerglings（輕型近戰單位）和 zealots + dragoons（中型近戰單位 + 中型遠距單位），分別又對應到單位數量 4、8、16、32、50。

透過實驗，本研究有以下結論：

UCTCD 使用的評估函式中，HPD 的表現最好。HPD 基本上只是雙方單位總生命值的差值，而 LTD 和 LTD2 則是試圖加入其他更多被視為重要的資訊，以此例來說就是單位所能造成的傷害。單位傷害確實是重要資訊，但這個結果顯示，如此直接將單位傷害與單位生命值相乘，並不能提供更準確的狀態評估。

UCTCD 在純遠距單位的狀況下，強度顯著的超越內建 AI；但在有近戰單位參與的情況下，強度就不及內建 AI。問題明顯是來自於，模擬器為了顧慮模擬效能，忽略了單位碰撞，這讓近戰單位無法在實際上會產生大量碰撞的戰鬥中，擬定適合的策略。因為從模擬器的角度來說，單位在模擬時是可以彼此重疊的，無法做出足夠精準的估計。

UCTCD 在單位數量 16 以下的強度較高，數量再往上增加就會降低強度。這是因為，一旦單位數量增加，整體的狀態空間（state space）也會跟著膨脹，一方面造成模擬器做模擬時效能下降，讓總模擬次數減少，另一方面也讓我們能夠涵蓋到的搜尋空間，比例上相對於全部可能性來說是狹隘的。

然而，不管是為了讓模擬器處理單位碰撞，還是想讓適用的單位數量增加，這些都可以藉由硬體效能的改善來解決。畢竟本次實驗所使用的硬體，只是一般個人電腦的等級（Intel(R) Core(TM) i5-4460 CPU @ 3.2 GHz、DDR3 RAM 8 GB），就能有這樣的成果，換作是對於運算資源豐富的企業來說，像是 Google，要用上 MCTS 就沒有這方面的問題，

也能預期會有好上許多的結果。多的運算資源，我會建議先加強 exploitation，而非 exploration，因為好的子節點已經以 script 篩到前面了，其實最佳選擇就在前面的機率相對較高，越後面的子節點，也就是跟 script 產生出來的，越不像的那些，就越不可能是好的子節點。某種程度上，exploration constant 的實驗結果是 0.2，也佐證了這一點。

我們也能藉由改進產生新動作組（actions）的品質，來讓 UCTCD 的效果更好。本研究某種程度上也將那些，已經在即時戰略中被證實有效的腳本 NOKAV、Kiting-AV，整合進 UCTCD，來讓搜尋更具有方向性，而不是在廣大的狀態空間中很發散的隨機搜尋，但除了這兩個腳本產生的動作以外，之後的動作都是根據它們來做隨機變形，這個部份或許還能加入更精緻的作法。即使這會增加對於專家知識的需求，也是一個值得嘗試的取徑。

而這個研究成果，同時也適用於任何具有單位控制（unit control）元素的即時戰略遊戲，以及其相關變形或混合。例如在 2016 年 1 月由 *Supercell* 推出的手機遊戲 *Clash Royale*[17]，它在 4 月就被預期即將為遊戲團隊帶來十億收入[18]。它結合了許多不同遊戲類型的要素，其中一個就是即時戰略中的單位控制，但為了方便在手機上遊玩，玩家所能決定的事情就簡化為，在合適的時間將單位放置在合適的地點，然後遊戲會接管這個單位，依照固定的規則去移動和攻擊。

以 *Clash Royale* 這樣的遊戲來說，我認為要將 MCTS 應用在它的單位控制上，相對於傳統的即時戰略遊戲來說，是更為容易的。因為，遊戲中會產生不確定性的機會變少了，一旦單位被放置在場面上，它的一切動作都是可預測的。不像本次研究中使用的 *StarCraft*，能夠適時為任一單位挑選新的攻擊對象。理論上，只要同樣的將 UCTCD 的概念融入到遊戲規則中，應該就能產生出不錯的效果。

對遊戲開發者來說，這個研究成果，也能為相關即時戰略遊戲 AI 的開發有所助益。一般業界遊戲所發展出來的人工智慧，真正在追求的是，要以有趣的方式故意輸給玩家。(Lidén 2003)然而，絕大多數的遊戲所使用的人工智慧技術，都還停留在很早期的階段，像是腳本（scripting）、有限狀態機（finite state machines）、決策樹（decision trees）和基於規則系統（rule-based system），這使得遊戲中人工智慧的行為往往是可預測的（predictable）、無調適性的（non-adaptive）、不彈性的（inflexible）。(Ontanon 2012, Robertson and Watson 2014)

為了減少對專家知識的依賴，又要有足夠的彈性讓玩家保持新鮮感，MCTS 這時就成了一個很好的選擇。之所以會特別強調遊戲開發者，在於 MCTS 需要一個可靠的模擬器，而這個需求對開發者而言不會是太困難的事情，因為所有的遊戲機制都是可取得的，不像我們研究者在使用商業遊戲的時候，都只能藉由臆測和估計。

---

[17] Clash Royale Official Site: https://clashroyale.com/

[18] 2016/04/14 “Supercell’s Clash Royale on pace for $1B year after $110M debut last month” http://venturebeat.com/2016/04/14/supercells-clash-royale-on-pace-for-1b-year-after-110m-debut-last-month

總結來說，這次研究將 MCTS 的可能性，拓展到即時戰略遊戲的單位控制上，證明它是個具有發展潛力的方法，只是更多方法上的調整和改進，還需要往後的研究來做其他嘗試，才能有所突破。

從象棋到圍棋，再來是即時戰略，那接下來遊戲對局相關的人工智慧，要再往哪個遊戲發展呢？有學者認為，答案很可能是 *League of Legends*[19]。[20]這是一款五個玩家對五個玩家的 MOBA（Multiplayer Online Battle Arena）遊戲，也有人稱其為 ARTS（Action Real-Time Strategy）遊戲。和即時戰略遊戲最顯著的差別在於，玩家不再能控制己方陣營的所有單位了，而是只可以操作一個單位，和隊上的其他四個玩家合作，總共五人對抗敵方五人。這讓人工智慧多了一個任務是，該如何揣測隊友的策略，與他人一起行動，在戰鬥中協同施放技能呢？甚至，人工智慧能不能在隊友表現不好的時候，挺身而出，和隊友溝通，指揮全隊贏得勝利呢？這個問題非常值得解決。就像報導中 Michael Cook 所說的，「能夠戰勝人類是一回事，但真正成功的人工智慧，是要能夠和人類合作的。」

[19] League of Legends: https://leagueoflegends.com

[20] AlphaGo: beating humans is one thing but to really succeed AI must work with them | The Guardian https://www.theguardian.com/technology/2016/mar/15/alphago-ai-artificial-intelligence-beating-humans

參考文獻

Aha, D. W., et al. (2005). Learning to win: Case-based plan selection in a real-time strategy game. Case-based reasoning research and development, Springer: 5-20.

Balla, R.-K. and A. Fern (2009). UCT for Tactical Assault Planning in Real-Time Strategy Games. IJCAI.

Baumgarten, R., et al. (2008). "Combining ai methods for learning bots in a Real-Time Strategy Game." International Journal of Computer Games Technology **2009**.

Billings, D., et al. (1999). Using probabilistic knowledge and simulation to play poker. AAAI/IAAI.

Bouzy, B. and B. Helmstetter (2004). Monte-carlo go developments. Advances in computer games, Springer: 159-174.

Buro, M. (2003). Real-time strategy games: A new AI research challenge. IJCAI.

Cadena, P. and L. Garrido (2011). Fuzzy case-based reasoning for managing strategic and tactical reasoning in starcraft. Advances in Artificial Intelligence, Springer: 113-124.

Chung, M., et al. (2005). "Monte Carlo planning in RTS games." CIG.

Churchill, D. and M. Buro (2011). Build Order Optimization in StarCraft. AIIDE.

Churchill, D. and M. Buro (2012). Incorporating Search Algorithms into RTS Game Agents. Eighth Artificial Intelligence and Interactive Digital Entertainment Conference.

Churchill, D. and M. Buro (2013). Portfolio greedy search and simulation for large-scale combat in starcraft. Computational Intelligence in Games (CIG), 2013 IEEE Conference on, IEEE.

Churchill, D., et al. (2012). "Fast Heuristic Search for RTS Game Combat Scenarios."

Cowling, P., et al. (2012). "Ensemble determinization in monte carlo tree search for the imperfect information card game magic: The gathering." Computational Intelligence and AI in

Games, IEEE Transactions on **4**(4): 241-257.

Gabriel, I., et al. (2012). Neuroevolution based multi-agent system for micromanagement in real-time strategy games. Proceedings of the Fifth Balkan Conference in Informatics, ACM.

Ginsberg, M. L. (1999). GIB: Steps toward an expert-level bridge-playing program. IJCAI, Citeseer.

Hagelbäck, J. and S. J. Johansson (2008). "The Rise of Potential Fields in Real Time Strategy Bots." AIIDE **8**: 42-47.

Jaidee, U., et al. (2011). Case-based learning in goal-driven autonomy agents for real-time strategy combat tasks. Proceedings of the ICCBR Workshop on Computer Games.

Judah, K., et al. (2010). Reinforcement Learning Via Practice and Critique Advice. AAAI.

Kabanza, F., et al. (2010). "Opponent Behaviour Recognition for Real-Time Strategy Games." Plan, Activity, and Intent Recognition **10**: 05.

Laird, J. and M. VanLent (2001). "Human-level AI's killer application: Interactive computer games." AI Magazine **22**(2): 15.

Lidén, L. (2003). "Artificial stupidity: The art of intentional mistakes." AI Game Programming Wisdom **2**: 41-48.

Marthi, B., et al. (2005). Concurrent hierarchical reinforcement learning. IJCAI.

Mishra, K., et al. (2008). Situation assessment for plan retrieval in real-time strategy games. Advances in Case-Based Reasoning, Springer**:** 355-369.

Muñoz-Avila, H. and D. Aha (2004). On the role of explanation for hierarchical case-based planning in real-time strategy games. Proceedings of ECCBR-04 Workshop on Explanations in CBR, Citeseer.

Ontañón, S., et al. (2007). Case-based planning and execution for real-time strategy games. Case-Based Reasoning Research and Development, Springer**:** 164-178.

Ontanon, S. (2012). Case acquisition strategies for case-based reasoning in real-time strategy

games. Twenty-Fifth International FLAIRS Conference.

Palma, R., et al. (2011). Combining expert knowledge and learning from demonstration in real-time strategy games. Case-Based Reasoning Research and Development, Springer**:** 181-195.

Perkins, L. (2010). "Terrain Analysis in Real-Time Strategy Games: An Integrated Approach to Choke Point Detection and Region Decomposition." AIIDE **10**: 168-173.

Robertson, G. and I. Watson (2014). "A review of real-time strategy game AI." AI Magazine **35**(4): 75-204.

Sailer, F., et al. (2007). Adversarial planning through strategy simulation. Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on, IEEE.

Schaeffer, J. (2001). "A gamut of games." AI Magazine **22**(3): 29.

Shantia, A., et al. (2011). Connectionist reinforcement learning for intelligent unit micro management in starcraft. Neural Networks (IJCNN), The 2011 International Joint Conference on, IEEE.

Synnaeve, G. and P. Bessiere (2011). A Bayesian model for RTS units control applied to StarCraft. Computational Intelligence and Games (CIG), 2011 IEEE Conference on, IEEE.

Szczepański, T. and A. Aamodt (2008). "Case-based reasoning for improved micromanagement in Real-time strategy games." Project Report NTNU-IDI.

Tesauro, G. (1995). "Temporal difference learning and TD-Gammon." Communications of the ACM **38**(3): 58-68.

Uriarte, A. and S. Ontanón (2012). Kiting in rts games using influence maps. Eighth Artificial Intelligence and Interactive Digital Entertainment Conference.

Weber, B. G., et al. (2010). Applying Goal-Driven Autonomy to StarCraft. AIIDE.

Weber, B. G., et al. (2011). A Particle Model for State Estimation in Real-Time Strategy Games. AIIDE.

Weber, B. G., et al. (2012). Learning from Demonstration for Goal-Driven Autonomy. AAAI.

Weber, B. G., et al. (2010). Reactive planning idioms for multi-scale game AI. Computational Intelligence and Games (CIG), 2010 IEEE Symposium on, IEEE.

Weber, B. G. and S. Ontanón (2010). Using automated replay annotation for case-based planning in games. ICCBR 2010 workshop on CBR for Computer Games.

Wintermute, S., et al. (2007). "SORTS: A human-level approach to real-time strategy AI." Ann Arbor **1001**(48): 109-2121.

Zhe, W., et al. (2012). Using Monte-Carlo Planning for Micro-Management in StarCraft. Proc. of the 4th Annual Asian GAME-ON Conference on Simulation and AI in Computer Games (GAMEON ASIA).