Homework assignment#1 (Chap3)

106971001 林上人

(A) Pseudo codes documentation

Pages: 3

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

①  state ← UPDATE-STATE(state, percept)
   if seq is empty then
②      goal ← FORMULATE-GOAL(state)
       problem ← FORMULATE-PROBLEM(state, goal)
③      seq ← SEARCH(problem)
       if seq = failure then return a null action
④  action ← FIRST(seq)
   seq ← REST(seq)
   return action
```

簡單的問題求解 Agent 設計

一開始 persistent 的部分先說明後並定義後面函數會用到的變數名詞，類似全域變數，包含 seq 用來存放動作序列，state 為對目前的狀態描述，goal 表示欲達成的目標，problem 則代表問題的描述，意指在給定目標的情況下要考慮哪些狀態和動作的過程。

整個流程包含三個部分，分別為 Formulate、Search、Execute，一開始①根據狀態和感知更新自己的狀態,後檢查動作序列是否為空，若不是空的表示目前其實是在一個解決問題的動作過程中,若爲空則進入②(Formulate)的階段,並制定目標和界定問題,然後在③(Search)尋找解決問題的動作序列,最後進入④(Execute)階段,從動作序列取出第一個動作後更新序列為剩餘的動作並將取出的動作回傳,function 每次回傳一個動作,直到動作序列為空才會再次進入②(Formulate) 、③(Search)階段。

Pages: 47

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
①   node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
②       if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        add node.STATE to explored
③       for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

BFS 演算法 - 先檢查最淺節點

此處使用 BFS 演算法來解決問題，①檢查問題的初始節點是否已經能夠通過 GOAL-TEST,通過就回傳 Solution，否則初始節點就為第一個 frontier 佇列元素，並利用 FIFO 的佇列結構實作 frontier 以達到 BFS 先檢查最淺節點的特性，explored 集合用來存放已經遍歷過且不屬於 frontier 的節點,後續的步驟以迴圈執行，②首先檢查 frontier 佇列內還有沒有元素 ,沒有則表示找不到解就回傳 failure，否則就從 frontier 佇列中 POP 出一個節點，因爲放在佇列中的節點都是沒有通過 GOAL-TEST，所以取出後也直接把它加入 explored 集合，然後③針對取出的節點找出子節點並檢查 ,如果子節點不屬於 explored 集合 ,也不屬於 frontier 佇列，就先對它做 GOAL-TEST,若通過則表示找到解回傳 Solution,沒通過就加入 frontier 佇列之中，整個迴圈會執行到 frontier 為空回傳 failure（②）或 child 通過 GOAL-TEST 回傳 Solution（③）爲止。

Pages: 61

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

① *node* ← a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0
① *frontier* ← a priority queue ordered by PATH-COST, with *node* as the only element
① *explored* ← an empty set
loop do
② **if** EMPTY?(*frontier*) **then return** failure
② *node* ← POP(*frontier*) /* chooses the lowest-cost node in *frontier* */
② **if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
② add *node*.STATE to *explored* Goal test after pop
for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
③ *child* ← CHILD-NODE(*problem*, *node*, *action*)
③ **if** *child*.STATE is not in *explored* or *frontier* **then**
③ *frontier* ← INSERT(*child*, *frontier*)
④ **else if** *child*.STATE is in *frontier* with higher PATH-COST **then**
④ replace that *frontier* node with *child*

UCS 演算法 – 改良 BFS,考慮路徑成本

演算法類似 BFS 演算法，但①不檢查問題的初始節點而是初始節點直接就為第一個 frontier 佇列元素，並把 frontier 佇列改成以 PATH-COST 排列的優先權佇列，explored 集合一樣存放走過的節點,後續的步驟以迴圈執行，②首先檢查 frontier 佇列內還有沒有元素，沒有則表示找不到解就回傳 failure，否則就從 frontier 佇列中 POP 出 PATH-COST 最低的節點，對節點做 GOAL-TEST，通過就回傳 Solution，否則把它加入 explored 集合，然後③針對取出的節點找出子節點並檢查，如果子節點不屬於 explored 集合，也不屬於 frontier 佇列，就依 PATH-COST 加入 frontier 佇列之中，但是④如果子節點已經存在於 frontier 佇列中，就表示有其他路徑已經到達過此節點，此時就必須比較到達的 PATH-COST，如果新的 PATH-COST 比較低就用這個子節點替換 frontier 佇列中的節點資料，因此即使目標節點已經先走到過也會在 frontier 佇列中不斷更新成最低成本的路徑，整個迴圈會因為無解而回傳 failure 或是執行到目標節點成為 frontier 佇列中成本最低的節點時通過 POP 和 GOAL-TEST 回傳 Solution 爲止（②）。

Pages: 85

```
①  function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
     return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

   function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
②    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
③    else if limit = 0 then return cutoff
     else
         cutoff_occurred? ← false
         for each action in problem.ACTIONS(node.STATE) do
④            child ← CHILD-NODE(problem, node, action)
             result ← RECURSIVE-DLS(child, problem, limit − 1)
             if result = cutoff then cutoff_occurred? ← true
             else if result ≠ failure then return result
⑤        if cutoff_occurred? then return cutoff else return failure
```

Depth limited search – 改良原 DFS, 限制深度 limit

此處以遞迴的方式實作 Depth limited search，①Depth limited 函式將初始節點和 limit 作為參數呼叫遞迴函式 Recursive-DLS，而 Recursive-DLS 函式內則②先檢查傳入的參數節點是否可以通過 GOAL-TEST，可以就回傳 Solution，或③傳入的 limit 等於 0 則回傳 cutoff，意指已經到達指定的深度，如果都不是則先宣告變數 cutoff_occurred 等於 false，然後④取出參數節點的子節點，並將子節點和深度-1 作為參數呼叫自己，直到遞迴跑完可以取得 result，因為 Recursive-DLS 有三種回傳值，所以 result 有三種可能分別為 Solution、failure、cutoff，檢查 result 若為 cutoff 表示已經到達指定深度但是還未找到解，修改 cutoff_occurred 值為 true，如果 result 不是 failure 則代表是 Solution，直接回傳 result，最後⑤檢查 cutoff_occurred 值若是 true 則回傳 cutoff，若是還未達到指定深度但是找不到解就回傳 false，這種可能就是子樹的規模比指定的深度小，而 Depth limited 函式的結果就和 Recursive-DLS 一樣分為找到解:Solution、未找到解但完成深度 limit 的搜尋:cutoff、未找到解且搜尋深度沒有達到 limit:failure。

Pages: 137

```
①  function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
       return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)
                                                  cutoff
   function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
②    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     successors ← [ ]
③    for each action in problem.ACTIONS(node.STATE) do
         add CHILD-NODE(problem, node, action) into successors
     if successors is empty then return failure, ∞
④    for each s in successors do /* update f with value from previous search, if any */
         s.f ← max(s.g + s.h, node.f))
     loop do
         best ← the lowest f-value node in successors
         if best.f > f_limit then return failure, best.f
⑤       alternative ← the second-lowest f-value among successors
         result, best.f ← RBFS(problem, best, min(f_limit, alternative))
         if result ≠ failure then return result        [parents]  [sibling]
```

RBFS 演算法 – linear space 和遞迴的類 standard best-first search

遞迴的方式實作 best-first search，①以初始節點和f_limit = ∞作為參數呼叫遞迴函式 RBFS，RBFS②首先檢查傳入的參數節點是否能通過 GOAL-TEST，否則③創建一個 successors 陣列並放入所有的子節點，如果沒有子節點就表示現在這個節點已經沒有路可以走向目標節點，就回傳 failure，接著④利用參數 node 更新 successors 的 f 值，意思是若此節點曾經被展開過，則節點的 f-value 應該要是 successor 的 lower-bound，避免冗餘動作造成效率不佳的問題，所以取 max(s.g + s.h, node.f)作為 successor 的 f-value，但是如果這個節點沒有展開過則 successor 的 f-value 就等於自己的成本估計值(s.g + s.h)，之後以迴圈方式執行後續動作，⑤先取得 f-value 最低的子節點與 f_limit 相比，因為 f_limit 記錄的是祖先節點最好的估計代價，所以如果 f_limit 比較小則表示有其他路較佳，因此僅回傳 failure 和此路徑知道的最小估計代價，儲存的 successors 陣列不保留以節省記憶體空間，如果 f-value 比較小則拓展這個子節點，以 parents 節點或 sibling 節點中較小的 f-value 作爲新的 f_limit 參數呼叫遞迴函式 RBFS，RBFS 的回傳值有兩種可能，一個是 Solution 就直接回傳，另一個則是 failure 和一個數值，而這個數值即代表這個節點下的 subtree 的估計代價會大於等於這個數值，因此我們就以這個值替換原本的 f-value,接著繼續跑迴圈直到找到目標節點。

Pages: 146

```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
①     if Queue is empty then return failure
②     n ← deepest least-f-cost node in Queue
      if GOAL-TEST(n) then return success
      s ← NEXT-SUCCESSOR(n)
③     if s is not a goal and is at maximum depth then
        f(s) ← ∞
      else
        f(s) ← MAX(f(n),g(s)+h(s))
④     if all of n's successors have been generated then
        update n's f-cost and those of its ancestors if necessary
⑤     if SUCCESSORS(n) all in memory then remove n from Queue
      if memory is full then
⑥       delete shallowest, highest-f-cost node in Queue
        remove it from its parent's successor list
        insert its parent on Queue if necessary
      insert s in Queue
  end
```

SMA*演算法 – 改善記憶體使用情形的 A*演算法

定義一個 f-cost 的優先權佇列 Queue 並放入初始節點，整個流程以迴圈持續執行，①檢查 Queue 是否為空，是的話就回傳 failure，然後②每次取出當前 Queue 中深度最深且 f-cost 最低的節點 n，並對 n 做 GOAL-TEST，若有通過就回傳 success，若沒有③就繼續往下尋找 n 的子節點 s，如果 s 不是目標節點而且已經達到搜尋的最大深度，我們就設定 s 的 f-cost 為∞，不然 f-cost 就是 n 的 f-cost 和 g(s)+h(s)中較大的值，④如果所有 n 的子節點都已經出現過，就可以依據子節點更新 n 和他的祖先節點的 f-cost 到最小的 f-cost，因為 f(n)=g(n) + h(n)，其中 g(n)的值可以被確實的算出來，⑤如果 n 的子節點都在記憶體中，我們可以暫時將 n 從 Queue 中移除，而⑥是 SMA*演算法中最重要的部分，如果記憶體滿了可以選擇將 f-cost 最高的節點移除，因為從當前情況看我們可視為從 f-cost 延伸的路徑都會有較高的 f-cost，藉此解省記憶體的使用，但 parent 必須紀錄被丟棄的部分中最低的 f-cost，以便最後仍然需要復原刪除的部分時可以使用，最後將 s 加入 Queue 中，持續直到找到目標節點為止。

Pages: 154

Effective Branching Factor Pseudo-code (Binary Search)

```
PROCEDURE EFFBRANCH (START, END, N, D, DELTA)
        COMMENT DELTA IS A SMALL POSITIVE NUMBER FOR ACCURACY OF RESULT.
①       MID := (START + END) / 2.
        IF (END - START < DELTA)
           THEN RETURN (MID).
        TEST := EFFPOLY (MID, D).
②       IF (TEST < N+1)
           THEN RETURN (EFFBRANCH (MID, END, N, D, DELTA) )
           ELSE RETURN (EFFBRANCH (START, MID, N, D, DELTA) ).
END EFFBRANCH.

PROCEDURE EFFPOLY (B, D)
③       ANSWER = 1.
        TEMP = 1.
        FOR I FROM 1 TO (D-1) DO
           TEMP := TEMP * B.
           ANSWER := ANSWER + TEMP.
        ENDDO.
        RETURN (ANSWER).
END EFFPOLY.

For binary search please see:  http://en.wikipedia.org/wiki/Binary_search_algorithm
An attractive alternative is to use Newton's Method (next lecture) to solve for the root  (i.e., f(b)=0) of
        f(b) = 1 + b + ... + b^d - (N+1)
```

使用 Binary Search 的方式計算 Effective Branching Factor

首先須知道 Effective Branching Factor(b^*)的定義，當 expanded 的節點總數為 N，解的深度為 d 時要符合

$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$

此處使用 binary search 的方式計算b^*，先估計一個b^*的值區間可能介於 START 和 END 之間，然後①取 START 和 END 的中間值作為b^*的猜測值b'，並丟入 EFFPOLY 計算，③利用猜測值b'和 d 計算 ANSWER = $N'+1 = 1 + b' + (b')^2 + \cdots + (b')^d$，並回傳計算結果，最後②把猜測值的計算結果$N'+1$與真實 N+1 做比較，若小於 N+1 表示$b'$低了，因此把 MID 到 END 作為新的區間再猜測新的值做計算，若大於 N+1 表示b'高了，因此把 START 到 MID 作為新的區間再猜測新的值做計算，持續執行到算出正確的b^*。

(B) Exercises

3.6 Consider the n-queens problem using the "efficient" incremental formulation given on page 72. Explain why the state space has at least $\sqrt[3]{n!}$ states and estimate the largest n for which exhaustive exploration is feasible. (*Hint*: Derive a lower bound on the branching factor by considering the maximum number of squares that a queen can attack in any column.)

Ans :

因皇后放置後最多可攻擊三格，所以當第一欄有 n 個位置可以放時，第二欄最少還會有 n-3 個位置可以放，依此類推可寫成

$S \geq n(n-3)(n-6)(n-9)(n-12)\cdots$

$\rightarrow S^3 \geq n \cdot n \cdot n \cdot (n-3)(n-3)(n-3)(n-6)\cdots$

$\rightarrow S^3 \geq n \cdot (n-1) \cdot (n-2) \cdot (n-3)(n-4)(n-5)(n-6)\cdots$

$\rightarrow S^3 \geq n!$

$\therefore S \geq \sqrt[3]{n!}$

State space 至少有 $\sqrt[3]{n!}$

3.15 Which of the following are true and which are false? Explain your answers.

a. Depth-first search always expands at least as many nodes as A* search with an admissible heuristic.

b. $h(n) = 0$ is an admissible heuristic for the 8-puzzle.

c. A* is of no use in robotics because percepts, states, and actions are continuous.

d. Breadth-first search is complete even if zero step costs are allowed.

e. Assume that a rook can move on a chessboard any number of squares in a straight line, vertically or horizontally, but cannot jump over other pieces. Manhattan distance is an admissible heuristic for the problem of moving the rook from square A to square B in the smallest number of moves.

Ans :

a. False : DFS 不考慮成本會一直往下搜尋，而 A*還考慮成本因此會拓展 f 低的路徑，若目標剛好在 DFS 不用回朔的路徑，拓展的路徑就可能會比 A*少。

b. True : 因為 0 已經是最小了，不可能高估。

c. False : 連續動作還是可以被離散化或概略化，所以還是可以使用。

d. True : 因為 BFS 不考慮成本，只要是 finite 就是 complete

e. False : 因為在直線上城堡在一步內移動的距離是不限格數的，所以 Manhattan distance 是有可能高估的

3.22 Prove each of the following statements, or give a counterexample:

a. Breadth-first search is a special case of uniform-cost search.

b. Depth-first search is a special case of best-first tree search.

c. Uniform-cost search is a special case of A* search.

Ans :

a. True : uniform-cost search 就是從 BFS 考慮路徑成本演變而來，所以當 cost 都相等時 uniform-cost search 就會表現的與 BFS 一樣

b. True : 當定義 best-first tree search 的 heuristic function 定義為 f(n) = -depth(n) 就可以表現得和 DFS 一樣

c. True : A*的 heuristic function : f(n) = g(n) + h(n) ，當定義 h(n)=0 時就會等於 Uniform-cost 的 heuristic function，他們的表現就會一樣

3.28 The **heuristic path algorithm** (Pohl, 1977) is a best-first search in which the evaluation function is $f(n) = (2 - w)g(n) + wh(n)$. For what values of w is this complete? For what values is it optimal, assuming that h is admissible? What kind of search does this perform for $w = 0$, $w = 1$, and $w = 2$?

Ans:

Uniform-cost search 和 A*是 complete, Greedy 不是 complete,

如果要符合 complete → $0 \leq w < 2$

Uniform-cost search 和 A*是 optimal,且 h 不能高估

如果要符合 optimal → $0 \leq w \leq 1$

w = 0 , f(n) = 2g(n) → Uniform-cost search

w = 1 , f(n) = g(n) + h(n) → A* search

w = 2 , f(n) = 2h(n) → Greedy best fisrt search

3.29 Consider the unbounded version of the regular 2D grid shown in Figure 3.9. The start state is at the origin, (0,0), and the goal state is at (x, y).

a. What is the branching factor b in this state space?
b. How many distinct states are there at depth k (for $k > 0$)?
c. What is the maximum number of nodes expanded by breadth-first tree search?
d. What is the maximum number of nodes expanded by breadth-first graph search?
e. Is $h = |u - x| + |v - y|$ an admissible heuristic for a state at (u, v)? Explain.
f. How many nodes are expanded by A* graph search using h?
g. Does h remain admissible if some links are removed?
h. Does h remain admissible if some links are added between nonadjacent states?

Ans:

a.因為是無邊界，所以每個點都可以向四方發展，分支因子為 4

b. dinstinct states，一個點就是一個 state，因為是 at depth = k 不是 when depth = k 所以我們計算 at depth = k 的 node 有 4k 個，所以 states 有 4k

c.因為樹的每一個 node 都有 4 個子節點，因此最差探索點數量發生在所有子節點都走一遍，假設 goal node 在 depth=k 上，則

節點數量 n = $1 + 4 + 4^2 + 4^3 + \cdots + 4^k$

$= \frac{1 \cdot (4^{k+1} - 1)}{4 - 1}$

又因 2D grid 產生是正方形，所以可知 k = |x|+|y|

→ nodes = $\frac{4^{(|x|+|y|+1)} - 1}{3}$

d. 因為是 graph，走過的點就不走，因此最差探索點數量發生在所有節點走一遍，假設 goal node 在 depth=k 上，則

節點數量 n = 1 + 4(1+2+3+4….k)

= 1 + 2(k+1)k

又因 2D grid 產生是正方形，所以可知 k = |x|+|y|

→ nodes = 1 + 2(|x| + |y| + 1)(|x| + |y|)

e.符合 admissible，因為每格邊距都是固定 1，所以不會高估

f. |x|*|y|個節點，因為使用 *h* 時在 origin(0,0)到目標(x,y)之間的路線都是最佳解

g.是，因為移除 links 可能讓最佳路線變長，但是用 *h* 估算路線還是不會高估所以符合 admissible

h.不會，因為加入新的節點在不相鄰的點之間可能讓最佳路線變短，因為 *h* 估算路線不包含對角線之類的所以可能高估，不符合 admissible

3.32 Prove that if a heuristic is consistent, it must be admissible. Construct an admissible heuristic that is not consistent.

Ans:

admissible : $h(n) \leqq k(n)$

consistent : $h(n) \leqq c(n, a, n') + h(n')$

Prove :

(1) 令 k(n)表示從 n 到目標的實際最低成本

(2) 令 $c(n, a, n')$ 表示從節點 n 經過動作 a 到達n'的成本

(3) 令 h(n) 表示從 n 到目標的估計成本

(4) 根據定義，若是 consistent 就會符合 $h(n) \leqq c(n, a, n') + h(n')$

(5) 假設 n 為 Goal，則$h(n) = 0 \leqq k(n)$ 成立

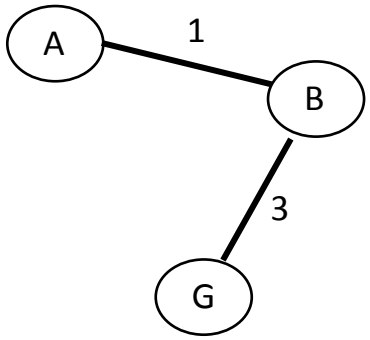
(6) 假設 n 不為 Goal 距離 Goal k 步 ,且n'為 Goal,則從 n 到n' 寫作

$h(n) \leqq c(n, a, n') + h(n')$

∵ (5)成立

∴ $h(n) \leqq c(n, a, n') + k(n') = k(n)$ 成立

可知若 heuristic 是 consistent 則他必定是 admissible



定義

k(x) = m

h(x) = 2 x (m/2)

h(A) = 4 ≦ k(A)

h(B) = 2 ≦ k(B) = 3

h(G) = 0 ≦ k(G)

∴ admissible

h(A) = 4 > 1 + h(B) = 3

∴ not consistent