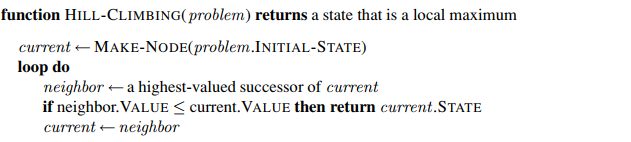
**Homework assignment#2 (Chap4)**

106971001 林上人

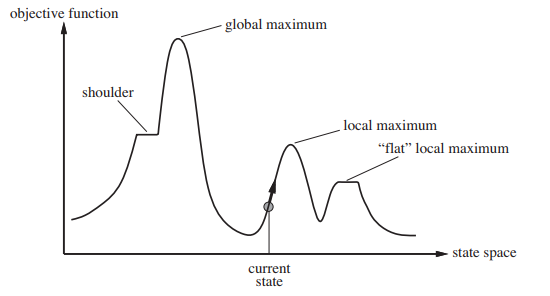
1. **Pseudo codes documentation**

Pages: 14



①

②

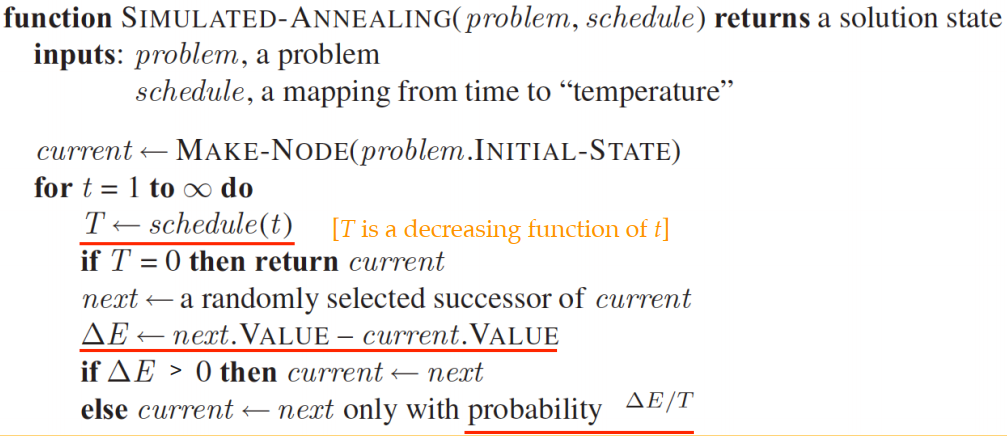


③

爬山演算法 - 尋找局部最優

爬山演算法是最基本的局部搜索演算法，首先①確定當前節點之後，②持續檢查successor，取出值最高的做比較，若比當前節點值低就回傳當前節點，否則就用值高的successor替換當前節點，以此方式不斷向值高的地方走，缺點就如上圖所示，雖然current state會一直向上走直到③local maximum處，找到局部最優，但是其實在另一個地方還存在更高的global maximum。

Pages: 28



③

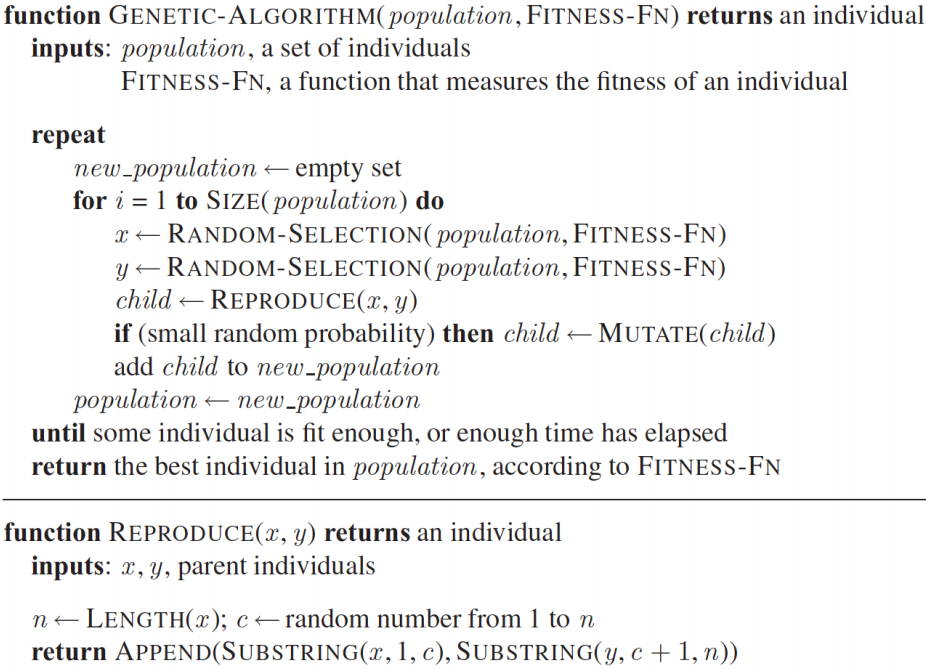
②

①

Simulated-Annealing演算法 – 可以下山的登山演算法

開頭①確定初始的當前節點後，我們用②schedule函數取得一個T值，這個T值會隨時間下降，當T降至0時就回傳當前的節點，而③處開始即是Simulated-Annealing實作可以下山的部分，首先取得successor的方式不再直接挑選值高的，而是以隨機的方式選取，若值比當前節點高則替換當前節點（上山），若值比當前節點低則以的機率決定是否替換當前節點(下山)，因爲T會隨時間下降，所以時間往後下山的機會就會持續降低，若T下降的足夠緩慢，找到global optimum的機率就越趨近1，但整個運作的時間就越久。

Pages: 43



⑤

④

②

③

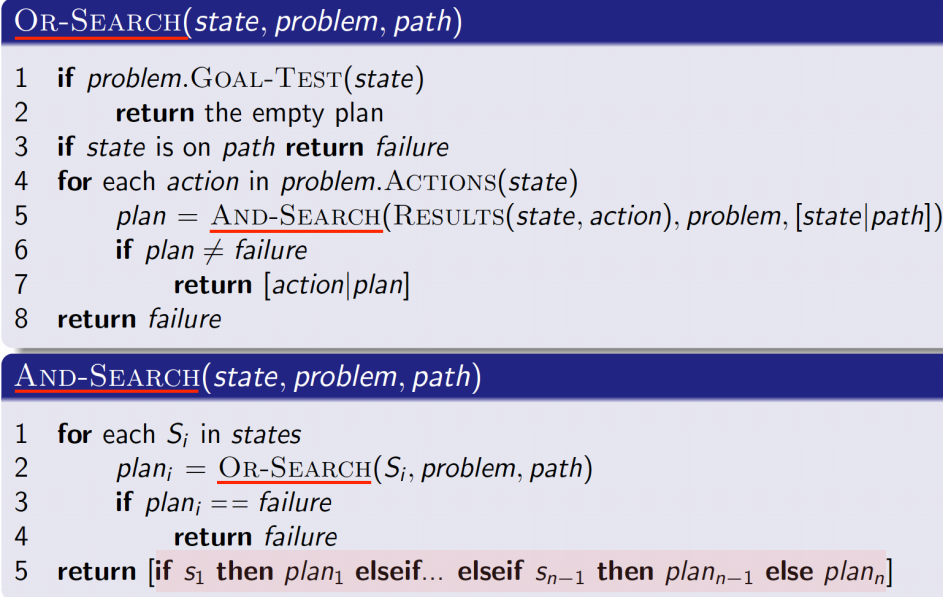
①

①

基因演算法

基因演算法流程首先①會輸入一個initial population，然後計算裡面個體的合適度，②從合適度在一定百分比以上的個體中，隨機選出兩個進行繁殖產生child，繁殖的方式如③，在總基因長度n中隨機選擇一個點c作為切割點，並由x的基因1~c和y的基因c~n組合成child，而產生的child④都會有一定的機率進行變異，並把child加入population集合之中，然後重複整個過程直到達成終止條件為止，此處終止條件⑤是設定為當出現某個個體的合適度已經足夠合適或是繁衍的時間已經足夠久就停止，並回傳population之中合適度最高的個體。

Pages: 48



③

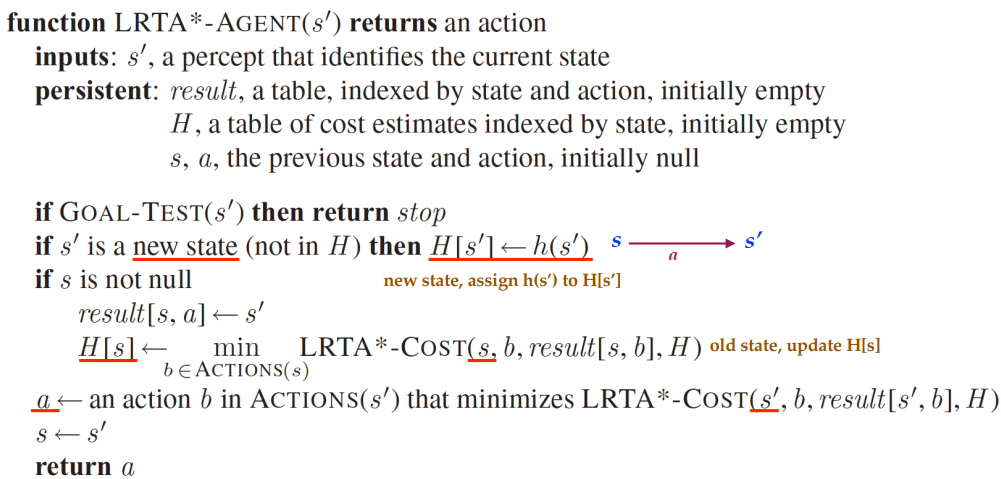
②

①

AND-OR 搜尋樹 – 求不確定性問題的可能解

OR節點選擇一個動作之後進行分支，因此某一分支後繼狀態有解即可，而AND節點則是由一個動作被選擇之後產生的所有狀態，因此必須處理所有的後繼狀態，才能找出解，因此在OR-SEARCH中①，檢查到達的是不是Goal，然後檢查state是否存在先前的路徑中，這部分是為了解決循環的情況，因為產生循環，所以如果有解從之前的路徑搜尋中就可以找到解，因此就把循環產生的部分回傳failure，然後②如前面所述，選擇一個動作之後到達AND節點，對這個AND節點做AND-SEARCH運算，③則表示AND-SEARCH中必須對所有產生的狀態都有策略規劃才能找到所有的解，並以if-than-else的方式回傳規劃。

Pages: 77

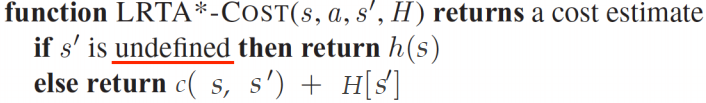


⑤

③

②

①

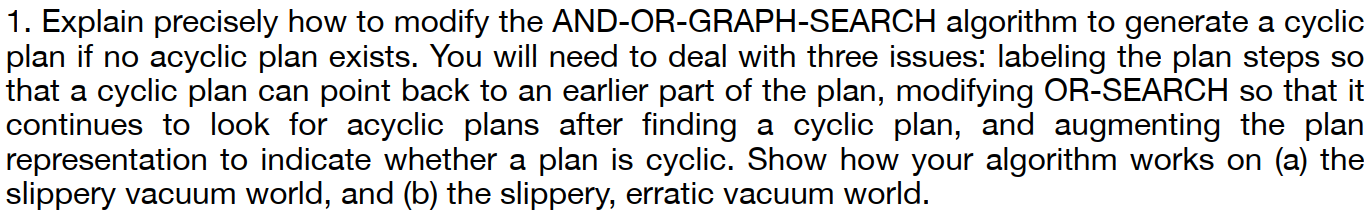


④

LRTA\*

本演算法初始建構元素，①處使用result建構環境轉移的地圖情形，H則記錄每個狀態的估計代價，首先取得下一個狀態時還是先對他做GOAL-TEST，之後②若新狀態未曾出現過在H中就先用A\*的h(s’)估計值代表，然後③使用result表紀錄狀態轉移的情況，並且使用LRTA\*-COST更新H表中H(s)的值為下一個動作的最小代價估計值，LRTA\*-COST做的事情如④所示，若新狀態是未曾出現過的就先以h(s)代表，否則就更新H表的值為從狀態S轉移到S’的成本加上S’的估計值，最後⑤則選出COST最小的action，並更新當前狀態S為S’。

1. **Exercises**



③

②

①

**function** OR-SEARCH(state, problem, path) **returns** a conditional plan, or failure

**if** problem.GOAL-TEST(state) **then return** the empty plan

**if** state is on path **then return** loop

cyclic - plan ← None

**for each** action **in** problem.ACTIONS(state) **do**

plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])

**if** plan ̸= failure **then**

**if** plan is acyclic **then return** [action | plan]

cyclic - plan ← [action | plan]

**if** cyclic - plan ̸= None **then return** cyclic - plan

**return** failure

**function** AND-SEARCH(states, problem, path) **returns** a conditional plan, or failure

loopy ← True

**for each** si **in** states **do**

plan i ← OR-SEARCH(si, problem, path)

**if** plan i = failure **then return** failure

**if** plan i ̸= loop **then** loopy ← False

**if** not loopy **then**

**return** [**if** s1 **then** plan1 **else if** s2 **then** plan 2 **else** . . . **if** sn-1 **then** plan n-1 **else** plan n]

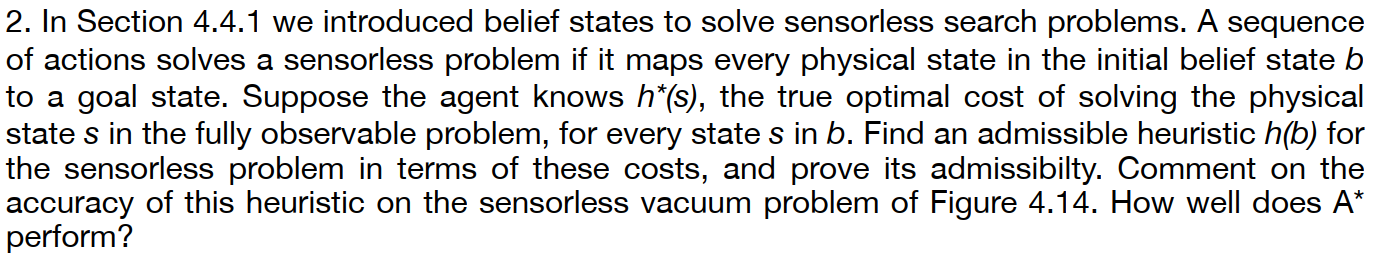
**return** failure

Deal with three issues:

**①** labeling the plan steps so that a cyclic plan can point back to an earlier part of the plan : 當發生路徑循環的時候就回傳loop表示要回到此狀態沿著路徑最近一次出現的時候

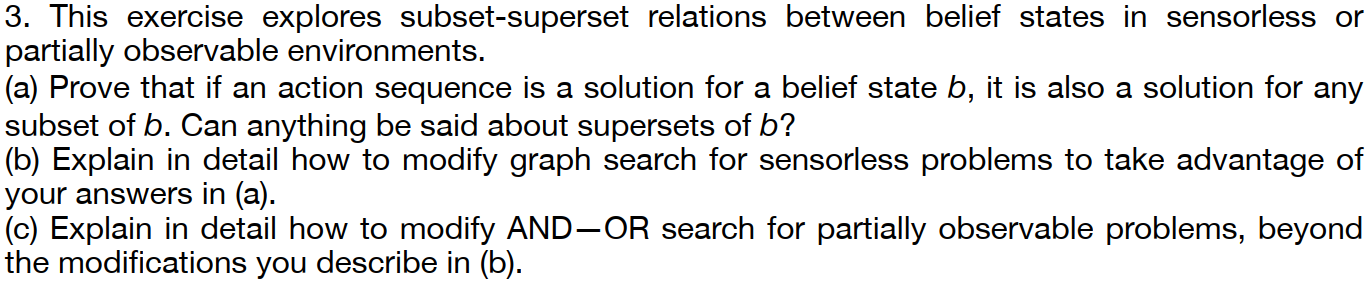
②③modifying OR-SEARCH so that it continues to look for acyclic plans after finding a cyclic plan, and augmenting the plan representation to indicate whether a plan is cyclic : 可以在plan中檢查是否存在loop，進而讓plan持續搜尋非循環路徑規劃

1. 因為加入回傳loop，可以讓slippery vacuum world發生動作失敗時，持續嘗試直到動作成功
2. labeling the plan steps回傳loop主要針對slippery world, And-Or-Search則針對erratic world



因為h\*(s)是true optimal cost，而我們需要找admissible heuristic h(b)，而根據admissible的定義h(b)必須不高估，也就是須小於等於實際成本所以可以定義一個belief-state的heuristic為所有在這個belief-state裡面的physical state之中h\*(s)最大的值 , s = belief state中的physical state

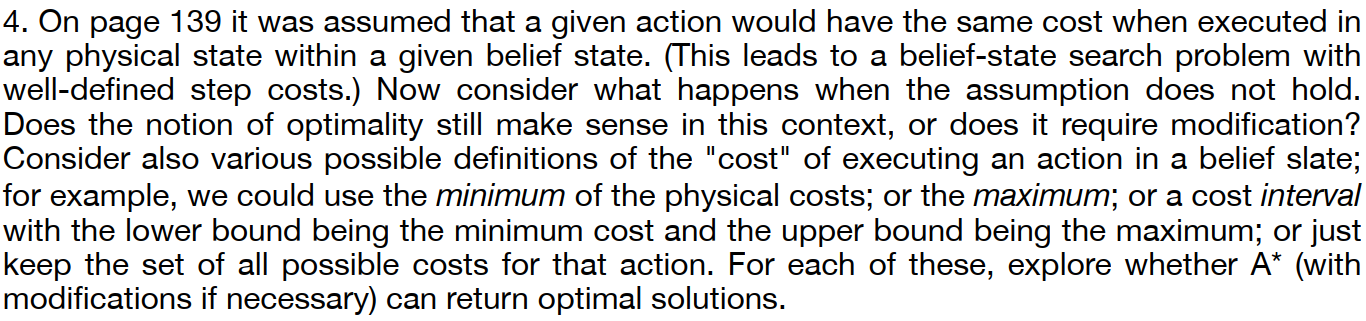
在Figure 4.14使用此估計法時，initial belief state和belief state{4,5,7,8}、{3,5,7}、{4,6,8}的h值會比實際上到達goal state的距離少1，因此A\*在到達goal state之前也會先拓展belief state{4,5,7,8}、{3,5,7}、{4,6,8}



**a**. 當一個動作序列為一個belief states *b*的解時，他會解決*b*所包含的所有狀態，所以*b*的子集合同樣能解，但是supersets就不行，因為supersets可能會包含不存在於*b*的狀態。

**b**. On expansion of a node, do not add to the frontier any child belief state which is a superset of a previously explored belief state.

**c**. If you keep a record of previously solved belief states, add a check to the start of ORsearch to check whether the belief state passed in is a subset of a previously solved belief state, returning the previous solution in case it is.



Consider a very simple example: an initial belief state {S1, S2}, actions a and b both  
leading to goal state G from either initial state

c(S1, a, G) = 3 ; c(S2, a, G) = 5 ;  
c(S1, b, G) = 2 ; c(S2, b, G) = 6 .  
In this case, the solution [a] costs 3 or 5, the solution [b] costs 2 or 6. Neither is “optimal” in  
any obvious sense. In some cases, there *will* be an optimal solution.

Let us consider just the deterministic case. For this case, we can think of the cost of a plan as a mapping from each initial physical state to the actual cost of executing the plan. In the example above, the cost for [a] is {S1:3, S2:5} and the cost for [b] is {S1:2, S2:6}.

We can say that plan p1 *weakly dominates* p2 if, for each initial state, the cost for p1 is no higher than the cost for p2. If a plan p weakly dominates all others, it is optimal. Notice that this definition reduces to ordinary optimality in the observable case where every belief state is a singleton.

As the preceding example shows, however, a problem may have no optimal solution in this sense. A perhaps acceptable version of A\* would be one that returns any solution that is not dominated by another.

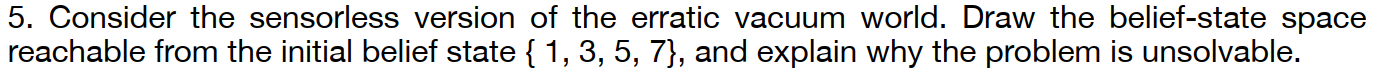
In particular, if we define the cost of a plan in belief-state space as the minimum cost of any physical realization, we violate Bellman’s principle. Modifying and extending the previous example, suppose that a and b reach S3 from S1 and S4 from S2, and then reach G from there:  
c(S1, a, S3) = 6 ; c(S2, a, S4) = 2 ;

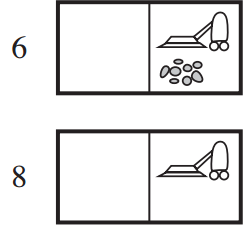
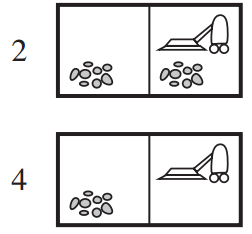
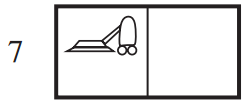
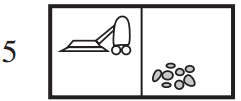
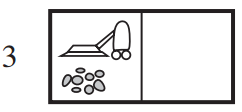
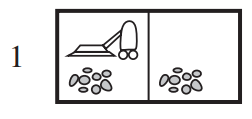
c(S1, b, S3) = 6 ; c(S2, b, S4) = 1 ;

c(S3, a, G) = 2 ; c(S4, a, G) = 2 ;

c(S3, b, G) = 1 ; c(S4, b, G) = 9 ;

In the belief state {S3, S4}, the minimum cost of [a] is min{2, 2} = 2 and the minimum cost  
of [b] is min{1, 9} = 1, so the optimal plan is [b]. In the initial belief state {S1, S2}, the four  
possible plans have the following costs:  
[a, a] : min{8, 4} = 4 ; [a, b] : min{7, 11} = 7 ; [b, a] : min{8, 3} = 3 ; [b, b] : min{7, 10} = 7 .  
Hence, the optimal plan in {S1, S2} is [b, a], which does *not* choose b in {S3, S4} even though that is the optimal plan at that point. This counterintuitive behavior is a direct consequence of choosing the minimum of the possible path costs as the performance measure.





Initial

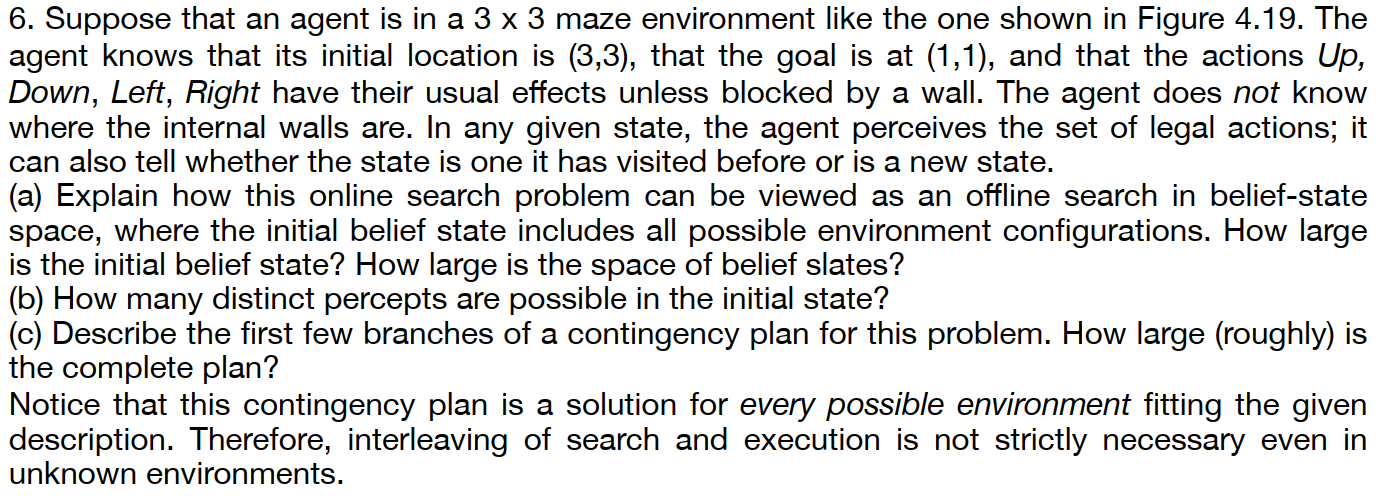
R

L

S

S

No solution is possible because no path leads to a belief state all of whose elements satisfy the goal.



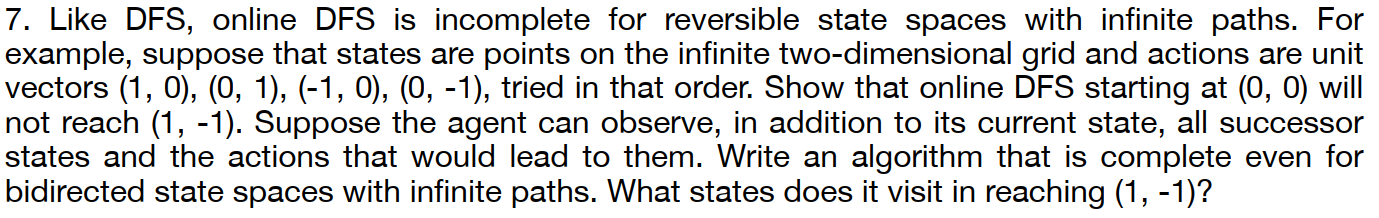
**a**. Online search is equivalent to offline search in belief-state space where each action in a belief-state can have multiple successor belief-states: one for each percept the agent could observe after the action. A successor belief-state is constructed by taking the previous belief-state, itself a set of states, replacing each state in this belief-state by the successor state under the action, and removing all successor states which are inconsistent with the percept.

The initial belief state has 1024 states in it, as we know whether two edges have walls or not but nothing more.

There are possible belief states, one for each set of environment configurations.

**b**. Assuming the external walls are known, there are two internal walls and hence 4 possible percepts.

**c**. From each belief state, the agent chooses a single action which can lead to up to 8 belief states. Given the possibility of having to retrace its steps at a dead end, the agent can explore the entire maze in no more than 18 steps, so the complete plan (expressed as a tree) has no more than nodes. On the other hand, there are just reachable belief states.



Infinite two-dimensional grid, so online DFS starting at (0, 0) will always apply vectors (1, 0) and will not reach (1, -1)

Since we can observe successor states, we always know how to backtrack from to a previous state. This means we can adapt iterative deepening search to solve this problem. The only difference is backtracking must be explicit, following the action which the agent can see leads to the previous state.  
The algorithm expands the following nodes:  
Depth 1: (0,0), (1,0), (0,0), (0,1), (0,0), (-1,0), (0,0), (0, -1), (0,0)

Depth 2: (0,0), (1,0), (2,0), (1,0), (1,1), (1,0), (1, -1)