國立清華大學資訊工程系 109 學年度上學期專題報告

專題名稱	Learning to route communication - limited vehicle fleet in road networks				
參加競賽或計畫	☐ 參加對外競賽	☐ 參與其他計畫	☐無參加對外競賽或任何計畫		
學號	106060015	106060018			
姓名	徐逢禧(組長)	趙柔茵(組員)			

摘要

在這個大數據時代中，數據可以發展出許多有趣且實用的應用。然而，大量的數據收集往往非常耗時。為了解決這個問題，本專題以 Reinforcement Learning 為方法，於路網上分析資料特徵並選擇最佳路徑，讓數據收集效率最大化。藉由在模擬環境中進行訓練和測試，來達到比起近似演算法為依據更加有效率的路徑選擇。

中華民國　　年　　月

一、專題研究動機與目的

真實世界的大量數據可以被用在許多和生活密切相關的應用上，例如搜救行動、巡邏、汙染檢測等。然而收集這樣大規模的數據是很費時的。本專題希望藉由一組數據收集單元（Data Collection Unit, DCU, or Agent）透過和彼此交換資訊，在區域內（路網）收集所需要的數據。

然而，若每個 agent 的路線沒有分配完善可能會耗費時間。並且我們要求 agent 要經過區域內所有路徑，不然數據有可能有偏差。

因此，本專題將路網中大規模的數據收集轉換成一個研究問題，即為路網中的 N 個 agent 規劃路線。題目的重要因素在於完整性（每條道路至少要有一個 agent 穿過進行數據收集）和效率（時間最小化）。我們為這個題目設計了一個近似演算法，即用此演算法可以在承受範圍內完成此任務。而目的是訓練 agent 在不同地圖中能利用 Reinforcement Learning 學習如何移動，並藉由 agent 之間互相溝通來達成比近似演算法更有效率的結果。

二、題目設計

(一) 題目介紹

我們將此題目的研究環境給定為一個無向路網，即 G=(V,E)。V 是頂點的集合、E 是道路的集合。假設有 N 個 agent，每個 agent 均有自己的起始點。目標是找到 N 組路徑，每一條路徑都對應到一個 agent，這些路徑的聯集可以覆蓋到所有的 E 集合（即整個路網），並將這些路徑加總起來的和盡可能最小化。

在我們給定的 input 中包含了頂點數量和其所在 x、y 座標、道路數量和長度及其連接的頂點、agent 數量和其起始位置及速度、agent 的溝通範圍。agent 要獲得關於地圖資訊的方式有兩種，第一種是親自經過該道路蒐集所需資料，第二種是透過與在溝通範圍內的其他 agent 進行資訊交換，我們定義如果兩個 agent 所在的 node 之間的距離(用座標算)在溝通範圍內，彼此就可以隨時進行溝通更新資料。假設某個 agent 正在走 edge(1,3)，如果 agent 位置未達道路的一半，我們定義 agent 此時所在的 node 為 node 1；如果 agent 位置超過道路的一半，我們定義 agent 此時所在的 node 為 node 3。

(二) 最終目標

此問題的最終目標為盡可能讓所有 agent 的路徑和最小化。由於各個 agent 速度不一定一樣，因此最終目標可等價於：使跑最快的 agent 的路徑

最小化，此篇報告之後提到的 cost 也以此做為計算。

三、研究方法與步驟

為了實作並證明出 reinforcement learning 在解決此問題上能勝過一般演算法，我們先設計出一個 Deterministic algorithm，在各個不同地圖上計算所需要的 cost，並以低於這些 cost 為目標，訓練出一個 DQN model，最後再互相比較。

在不同地圖的製作上，主要以 node 數量、根據 node 對應出的 edge 數量 (dense graph or sparse graph)、edge 長度及溝通範圍大小為變數做調整，而 agent 數量皆固定為 3 個。

以下分兩部分說明我們的研究方法與步驟。

第一部份說明 Deterministic algorithm 的設計方式，第二部分說明我們如何設計 DQN model 及我們在訓練 model 過程中使用的技巧跟方法。

(一) Deterministic algorithm

我們設計出的 deterministic 演算法有一個簡單的規則，就是讓 dcu 永遠只挑「走過次數最少」的 edge 走，如果有多條 edge 走過次數相等，便按照「固定的順序」挑 edge 走。

例如：agent A 在 node 1，而 node 1 有 3 條 edge 分別連到 node 2, node 3, node 4，而 edge(1,2)、edge(1,3)、edge(1,4)被走過的次數分別是(0,0,0)，因為該 3 條 edge 被走過次數相等，agent A 則照順序先挑 edge(1,2)走，此時該 3 條 edge 被走過的次數就更改成(1,0,0)。下次 agent A 再到 node 1 的時候，便會挑選 edge(1,3)走，因為該 edge 的走過次數最少且順序在 edge(1,4)前面，此時該 3 條 edge 被走過的次數就會更改成(1,1,0)。

按照此規則，直到所有的 edge 都被走過一次為止，並計算走最快的 agent 所走的距離當作 cost。

除此之外，我們根據 agent 獲取「各個 edge 走過次數」的方式細分成兩種演算法。第一種演算法，「Algo_WC」，是透過親自走過計算和 agent 之間彼此溝通獲得，而溝通的範圍則由地圖的原始設定來定義，此種稱為「無上帝視角演算法」。第二種演算法，「Algo」，則是直接把 agent 的溝通範圍設定成無限大，也就是說 agent 能隨時知道每一條 edge 被走過的次數，此種稱為「上帝視角演算法」。

普遍來說，「上帝視角演算法」的 cost 會較「無上帝視角演算法」小，因為它能讓 agent 獲得的資訊最完整，不會受到溝通範圍限制，因此在路徑選擇上會比較好。相關比較會在後面章節說明。

(二) DQN model and training techniques

RL 的核心理念為讓機器人(agent) 在看到身處環境(state) 後做出相應行動(action)，環境因此行動造成變化(next state)，我們對該行動給予一個評分(reward)，機器人事後便可以根據我們給的評分，調整下次要做出的行動。藉由這個機制，機器人在訓練過程中，會逐漸調整成做出能獲得「高 reward」的行動。

為了結合 RL，我們必須先找到與此問題相對應的 agent、state、action、next state、reward。Agent 為地圖上會移動的 3 個 agent；state 我們把它設定為一個矩陣，裡面存放著 agent 過去蒐集的資料，像是不同 edge 的長度、哪些 edge 被走過幾次……等等；action 則是對應到做路徑決策時，要選擇哪條可走的 edge；next state 是當 agent 做出 action 後原本的 state 出現的變化；reward 則是我們設定的一個 function，評斷 agent 該次的路徑選擇可以拿到幾分。

接下來，就是設計提供 agent 做路徑選擇的 DQN model 並進行 training，我們讓所有 agent 共用一個 model。

1. Basic introduction of our DQN model

   我們使用「pytorch」套件來實作 DQN model。總共有 11 層 layer，其中 1 層 input layer、9 層 hidden layer 和 1 層 output layer，除了 output layer 以外每一層皆使用 Relu 當作 activate function；loss function 使用 mean square error；optimizer 則是使用 Adam；learning rate 設為 0.002。下圖為 layer 的詳細內容，其中「nfeat」為我們挑選的 feature 數量(會在下面進行說明)。

```python
class DQN(nn.Module):
    def __init__(self, nfeat):
        super(DQN, self).__init__()
        self.L1 = nn.Linear(nfeat, 1024)
        self.L2 = nn.Linear(1024, 512)
        self.L3 = nn.Linear(512, 256)
        self.L4 = nn.Linear(256, 128)
        self.L5 = nn.Linear(128, 64)
        self.L6 = nn.Linear(64, 32)
        self.L7 = nn.Linear(32, 16)
        self.L8 = nn.Linear(16, 8)
        self.L9 = nn.Linear(8, 4)
        self.L10 = nn.Linear(4, 2)
        self.out = nn.Linear(2, 1)

    def forward(self, x):
        x = x.type(torch.FloatTensor)
        x = F.relu(self.L1(x))
        x = F.relu(self.L2(x))
        x = F.relu(self.L3(x))
        x = F.relu(self.L4(x))
        x = F.relu(self.L5(x))
        x = F.relu(self.L6(x))
        x = F.relu(self.L7(x))
        x = F.relu(self.L8(x))
        x = F.relu(self.L9(x))
        x = F.relu(self.L10(x))
        x = self.out(x)
        return x
```

圖 1

2. Input and output

DQN model 的 input 為一個矩陣，稱作 feature matrix，內容是我們為 edge 挑選的 feature，維度為「agent 所在 node 的 degree * feature 數量」。

例如：agent A 在 node 1，而 node 1 有 3 條 edge 分別連到 node 2, node 3, node 4，而 edge(1,2)、edge(1,3)、edge(1,4)的長度分別是(532, 166, 983)、此三條 edge 有連到的 edge 數量為(6, 2, 5)、被走過的次數分別是(0, 1, 2)，則作為 input 的 feature matrix 就等於：[[532, 6, 0], [166, 2, 1], [983, 5, 2]]。

由於地圖的資料只能透過 agent 自己蒐集或是跟別人溝通而來，所以如果有不知道的資料就會預設為 0。由此我們可以推斷出，在 agent 最一開始挑 edge 走的時候，作為 input 的 feature matrix 一定等於[[0, 0, 0], [0, 0, 0], [0, 0, 0]]。

Input 在經過 model 的 input layer 後，維度就會變成「agent 所在 node 的 degree * 1024」。再繼續下去直到 output layer 結束，就會變成一個「所在 node 連出去的 edge 數量 * 1」的矩陣，也就是我們 model 的 output。

3. Training process and techniques

Training 的一開始會初始一個 evaluation model 和一個 target model，兩者參數一樣，並讓 agent 使用 Epsilon-greedy policy 的方式透過 evaluation model 選擇 action。Epsilon 的初始值為 0.9，代表 agent 有 90%的機率從有連到的 edge 中隨機挑一個走，有 10%的機率從 evaluation model 的 output 中挑選擁有最高的值的 edge 去走。在訓練的過程中，epsilon 的值會慢慢減少直到 0.1 為止。目的是為了能增加 agent 探索不同路徑的機會，也避免在 training 的初期 agent 出現「繞圈圈」的情況。

此外我們使用 Temporal-difference update 的方式進行 training。在每一次 agent 做完路徑選擇後，該次的 action、state、next state、reward 就會成為一組樣本放進一個叫 replay buffer 的地方。之後再從 replay buffer 中隨機挑 32 組樣本當作一個 batch 進行 training。而 target model 每經過 10 次 training，就會把參數更新成 evaluation model 的參數。

至於用來 training 的地圖，我們分別用過 10、15、20 個 node 的地圖，後來發現成效並不會差很多，因此就挑 10 個 node 的地圖來 training，更省時間。做完 training 後的 model 便開始跑不同類型的地圖，之後再微調參數和變數重新進行 training。就時間上來看這樣做有槓桿上的效益，因為我們只用 10 個 node 的地圖 training 出來的 DQN model 卻能在不同 node 數量的地圖上跑，比如說 200 個 node 的地圖，然後效果維持一樣好。

4. Reward function

如何設定 reward function 是整個 training 過程中最重要的環節，它不只影響 DQN model 如何更新參數，也對 agent 的路徑選擇有決定性的作用。我們使用 reward shaping 的方法，先設定不同的 reward function 來得出不同 DQN model，並比較它們之間的 cost，接著微調 reward function，藉由重複這個過程來設計出更好的 reward function。

第一步是假設今天我們親自走這張地圖，會希望手邊有哪些資訊方便我們做更好的路徑選擇？並以此作為出發點，決定要讓 agent 蒐集和互相溝通哪些資訊。很直觀的來說，我們會希望一條 edge 被走過的次數越少越好，畢竟目的是盡快把每條 edge 都走過一遍，所以走重複的 edge 等同是浪費時間。除此之外，如果被迫要挑一條已經走過的 edge 走，我們希望 agent 能挑路徑比較短的 edge 走，這樣可以更快走完去走其他沒被走過的 edge。還有，如果能讓 agent 知道哪些 edge 有接到哪些 edge，或許能對地圖更熟

悉，藉此挑出更好的路徑。由此得知 agent 必須要蒐集和互相溝通的資訊就有「edge 被走過的次數」、「edge 的長度」及「edge 之間的相連資訊」。

第二步是設定 reward function。

(1) 我們會給 positive reward，當 agent 挑的 edge 沒被走過。意思是鼓勵這樣的行為。反之則給 negative reward，且它的絕對值跟 edge 被走過的次數呈正比。

(2) 我們會給一個 negative reward 且它的絕對值跟 agent 挑的 edge 的長度呈正比。意思是希望 agent 走比較短的路。

(3) 我們會給一個 negative reward 且它的絕對值跟目前所花的時間呈正比。意思是希望 agent 能越快走完整張地圖越好。

(4) 我們會給一個 negative reward 且它的絕對值跟 agent 要挑的 edge，它所連出去的 edge 中「有幾條 edge 沒被走過」呈反向關係。意思是希望 agent 所走的那條 edge，可以連到很多「沒被走過的 edge」，鼓勵 agent 探索新區域。

透過以上 4 種方法，我們設計兩個不同面向的 reward function，因此最終有兩個不同的 DQN model。

第一個 DQN model，「Model_dist」，的 reward function 依照(1)、(2)、(3)的規則做設計，預期讓 agent 學習到「不要走重複路徑」和「盡量走短的路」。

第二個 DQN model，「Model_newedgecon」，reward function 依照(1)、(4) 的規則做設計，預期讓 agent 學習到「不要走重複的 edge」和「走連到更多沒被走過的 edge 的 edge」。

四、效能評估與成果

以下將兩個「有、無上帝視角」的演算法(Algo、Algo_WC)、兩個不同的 DQN model (Model_dist、Model_newedgecon)，此四個方法針對不同地圖所需的 cost，與 Lower bound 進行比較，呈現最終的成果並討論。(Lower bound：走完地圖所需最低的 cost，算法是所有 edge 的長度加總，除以 agent 的數量。)

我們分成三種不同類型的地圖。第一種地圖是先設計一個 complete graph 然後隨機抽掉少量 edge；第二種地圖的 edge 數量只剩下 complete graph 的 edge 數量的一半；第三種地圖類似第一種地圖，但設定不同的 constraint (agent 的溝

通範圍)。此外，為避免地圖本身的設計造成結果偏差，所有結果皆是跑過同一類型的 10 張不同地圖後的結果取平均。

(一) 地圖類型一

表 1

Map information						
	Map01	Map02	Map03	Map04	Map05	Map06
Number of nodes	7	10	20	30	50	100
Number of edges	18	42	187	432	1222	4947
Constraint	500					
Cost						
Lower bound	3220.8	7270.8	33046.9	75349.9	215751.5	866940.4
Algo	4084.2	8530.2	37035	81414	228357.9	898777.8
Algo_WC	4069.8	8899.2	36574.2	81538.2	227181.6	897509.7
Model_newedgecon	4032	8361.9	34393.5	78280.2	221739.3	877149.9
Model_dist	4031.1	7972.2	34441.2	77458.5	219452.4	873639.9
Degree of improvement (with respect to “lower bound”)						
Model_newedgecon vs Algo	6.0%	13.4%	66.2%	51.7%	52.5%	67.9%
Model_dist vs Algo	6.2%	44.3%	65.0%	65.2%	70.6%	79.0%
Model_newedgecon vs Algo_WC	4.5%	33.0%	61.8%	52.6%	47.6%	66.6%
Model_dist vs Algo_WC	4.6%	56.9%	60.5%	65.9%	67.6%	78.1%

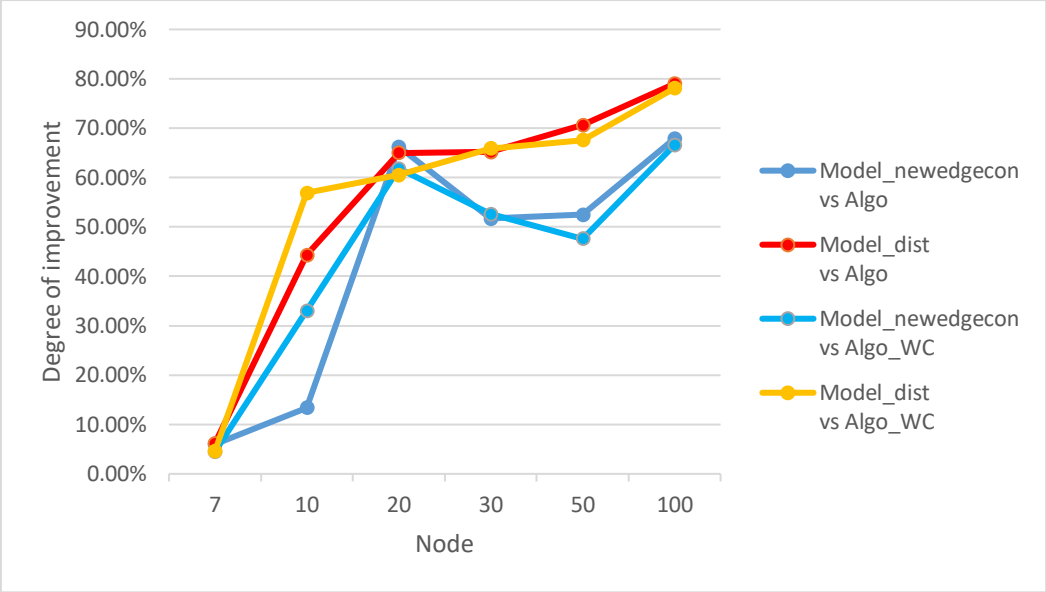

圖 2

從第一類型地圖的結果可分析出，node 越少的地圖，DQN model 和演算法的結果越相近。一旦將尺度拉到 20 個 node 以上，DQN model 的優勢就逐漸顯現出來了，省下的 cost 基本上都能超過 50%。普遍看起來，Model_dist 的表現略贏 Model_newedgecon。

這個情況是合理的，在 node 少的情況下 edge 也會跟著少，這時不小心走錯一條 edge 或多走一條較長的 edge，都會造成整體 cost 的明顯增加，因此運氣成份影響佔比較大。也因為地圖小資訊少，所以 DQN model 在還未累積足夠資訊做出比演算法更好的判斷時，就在前期先把地圖走完了。相反的，當 node 還有 edge 的數量變多，DQN model 就有足夠時間蒐集地圖資訊，並在中後期做出比演算法更好的路徑決策，少走一半以上的冤枉路。

(二)地圖類型二

表 2

Map information						
	Map01	Map02	Map03	Map04	Map05	Map06
Number of nodes	7	10	20	30	50	100
Number of edges	10	22	95	217	612	2475
Constraint	500					
Cost						
Lower bound	1659.2	3888	16222.9	37200.8	108911.8	434736.3
Algo	2498.4	5220	19533.6	41709.6	117707.4	460876.5
Algo_WC	2514.6	5261.4	18927	42847.2	118215.9	460686.6

Model_newedgecon	2374.2	4956.3	17849.7	39081.6	111931.2	441531.9
Model_dist	2545.2	4951.8	17894.7	39329.1	111989.7	440228.7
Degree of improvement (with respect to “lower bound”)						
Model_newedgecon vs Algo	14.8%	19.8%	50.9%	58.3%	65.7%	74.0%
Model_dist vs Algo	-5.6%	20.1%	49.5%	52.8%	65.0%	79.0%
Model_newedgecon vs Algo_WC	16.4%	22.2%	39.8%	66.7%	67.5%	73.8%
Model_dist vs Algo_WC	-3.6%	22.5%	38.2%	62.3%	66.9%	78.8%

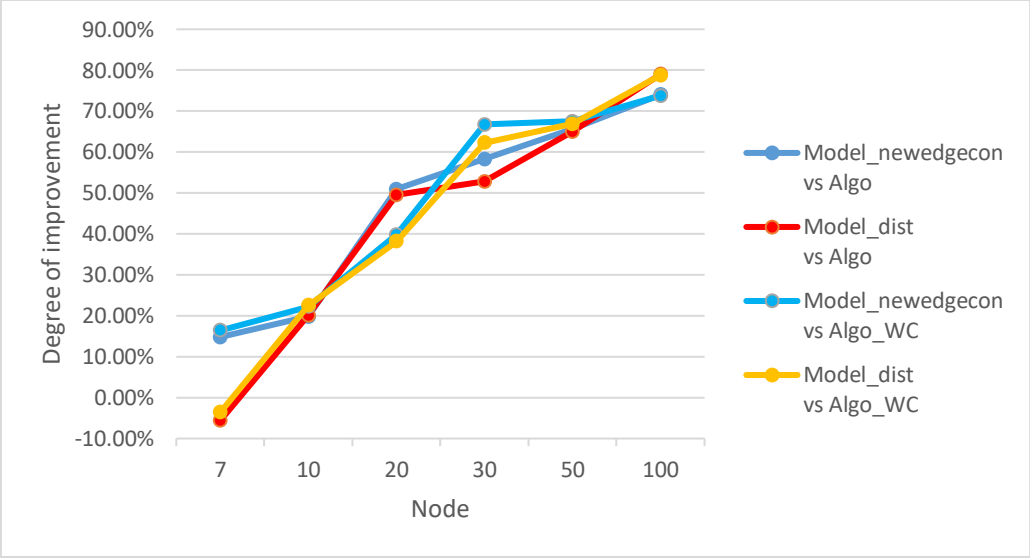

圖 3

從第二類型地圖的結果可分析出，node 越少的地圖，DQN model 和演算法的結果越相近，在有些地圖上甚至會輸給演算法。如果把尺度拉到 20 個 node 以上，DQN model 的優勢就逐漸顯現出來了，省下的 cost 基本上都能超過 40%，雖然比第一類型的地圖的 50%少，也算是不錯的成績。至於 Model_dist 的表現與 Model_newedgecon 則差不多。

隨著 node 和 edge 的數量增加，DQN mode 贏的更顯著，這跟第一類類型地圖的原因相仿。

(三) 地圖類型三

表 3

Map information						
	Map01	Map02	Map03	Map04	Map05	Map06

Number of nodes	30					
Number of edges	432					
Constraint	100	150	200	250	300	400
Cost						
Lower bound	75740.9	75740.9	75740.9	75740.9	75740.9	75740.9
Algo	81220.5	81220.5	81220.5	81220.5	81220.5	81220.5
Algo_WC	106302.6	92870.1	87042.6	84635.1	83068.2	81676.8
Model_newedgecon	90327.6	87205.5	82989	81432.9	79932.6	79293.6
Model_dist	94788.9	86573.7	84166.2	80553.6	79627.5	78490.8

表 4

Map information						
	Map07	Map08	Map09	Map10	Map11	Map12
Number of nodes	30					
Number of edges	432					
Constraint	500	600	700	800	900	1000
Cost						
Lower bound	75740.9	75740.9	75740.9	75740.9	75740.9	75740.9
Algo	81220.5	81220.5	81220.5	81220.5	81220.5	81220.5
Algo_WC	83021.4	82275.3	81603.9	81220.5	81220.5	81220.5
Model_newedgecon	78116.4	77779.8	77434.2	77601.6	77499.9	77499.9
Model_dist	77569.2	77409	77451.3	77388.3	77388.3	77388.3

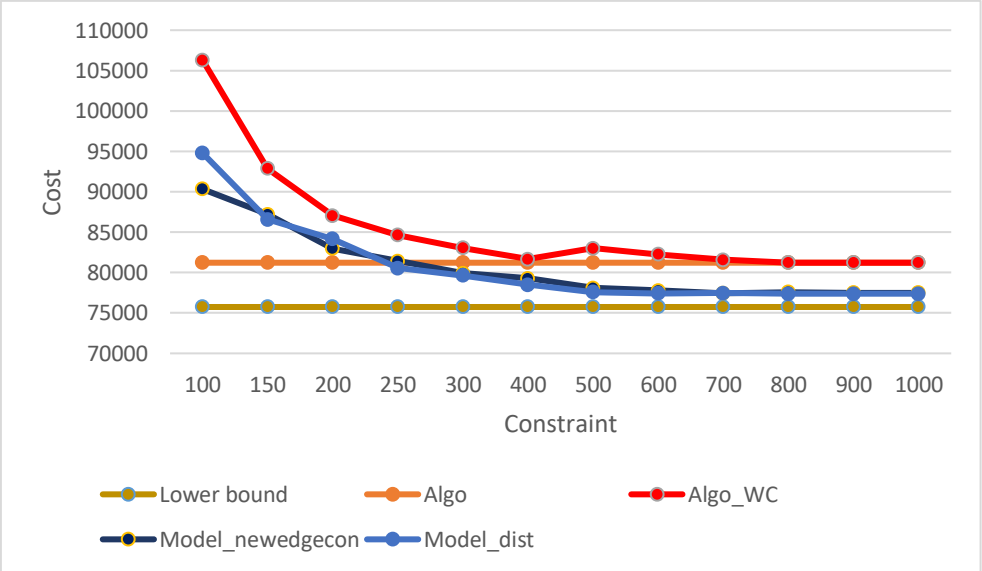

圖 4

從第三類型地圖的結果可分析出，constraint 小的地圖， Algo 都能贏過 DQN model 和 Algo_WC。隨著 constraint 逐漸變大，DQN model 的 cost 持續下降到最終比 Algo 還低，Algo_WC 的 cost 則逐較收斂到跟 Algo 一樣。

由此發現，除了 reward function 對成效影響重大外，溝通距離的大小也是很重要的因素。在題目的設計上，導致 agent 獲得的資訊有很大部分是來自跟其他人的溝通，若溝通距離過小，agent 的資訊也會過少，路徑決策就會明顯變差，輸給 Algo。

五、結論

從以上的各種比較結果可以得出，只要具有足夠的溝通距離，我們的兩個 DQN model 都可以在不同類型的地圖上勝過演算法，且省下的 cost 甚至能接近 80%！不管是有無上帝視角的演算法皆是。

至於兩個 DQN model 間的效能並沒有差太多，cost 的值大多相差都在幾條 edge 距離內。我們推測的原因是，雖然他們的 reward function 設計方向不一樣，但都有一個共同決定因子，是讓 agent 學習到「不要走重複的 edge」這件事，所以 reward 的最終決定權有可能被這項因子主導了，導致兩個 DQN model 的結果差別不大。但從另一方面來看，若 agent 只學習到「不要走重複的 edge」，那理論上 DQN model 的結果應該會跟演算法差不多(因為演算法就是根據這個作設計)，以此可證明，agent 還多學到了能輔助路徑決策東西，也就是我們 reward function 其他參數的設定。

最後，我們想這只是初步的 RL 結合 routing problem 研究，或許搭配不同的 RL 技巧、不同的 reward 設定方式……等等，教導 agent 學到更有用的路徑決策知識，讓 cost 更趨近於 lower bound，為此我們對未來的相關研究抱持樂觀的態度。

六、團隊合作方式

徐逢禧負責近似演算法的優化、GCN+RL 的實作、RL 的優化、動畫呈現。趙柔茵負責近似演算法的實作、RL 的實作、features 的設計。

七、參考文獻

1.https://github.com/tkipf/gcn
2.https://github.com/tkipf/pygcn/tree/master/pygcn
3.https://kknews.cc/zh-tw/code/yegrgnb.html