前言：

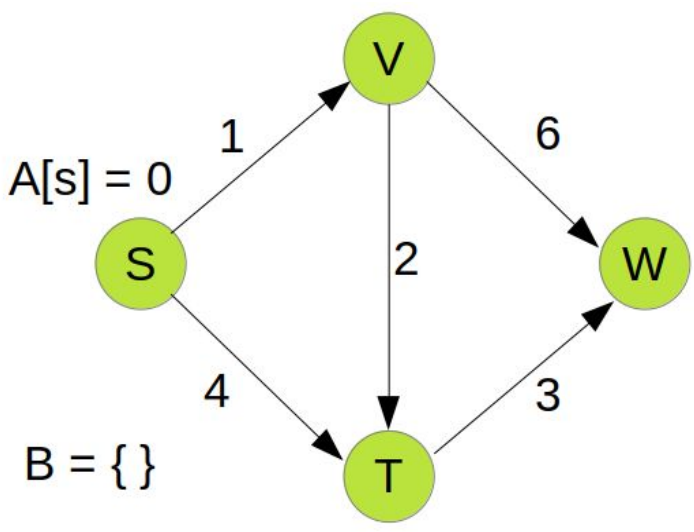
第二周的課程專注在 Dijkstra 最短路徑演算法，為了提高算法的效率，引入了重要的資料結構 Heap，講述 Heap 的基本性質與操作 (Heap 將由另文說明)。作業是利用 Dijkstra 算法求有向圖的最短路徑。

Dijkstra Shortest Path Algorithm：

我們知道 BFS 能找出最短路徑，那為什麼還需要其他的演算法來計算？ 現考慮一有向圖，且邊的權重等於 N (N≥1) ，BFS 則不保證能找到最短路徑。或許你會有疑問，那把所有邊依照權重，全部轉換成數個權重為 1 的邊不就行了？但這樣做圖可能會變得非常龐大，計算起來效率低落。Dijkstra 在 1956 年發現了 Dijkstra Shortest Path 演算法，利用 BFS 及貪婪演算法的概念，解決正權重有向圖的單源最短路徑問題。

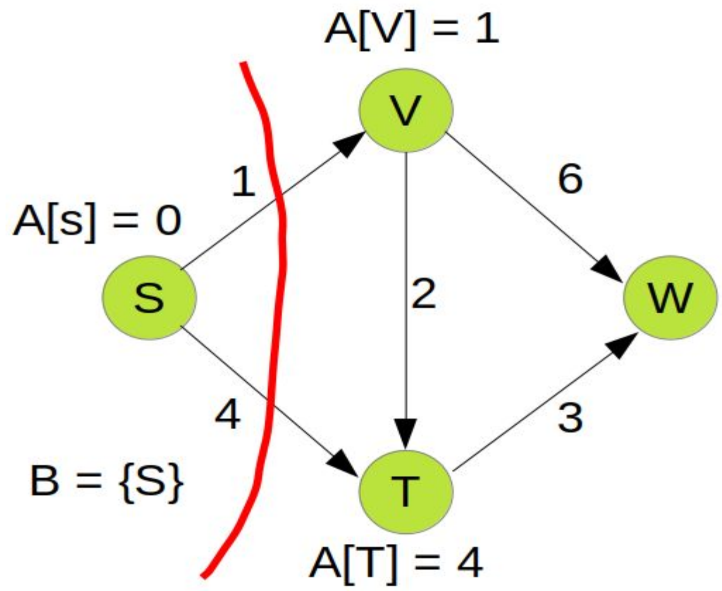
我們先用圖片來解說 Dijkstra 演算法的概念，假設我們要算 S → W 的最短路徑：

Step 1. 建立一個一維矩陣 A，存放各點到 S 的距離，此外建立一個集合 B，存放已遍歷的節點，將起點距離設為 0。

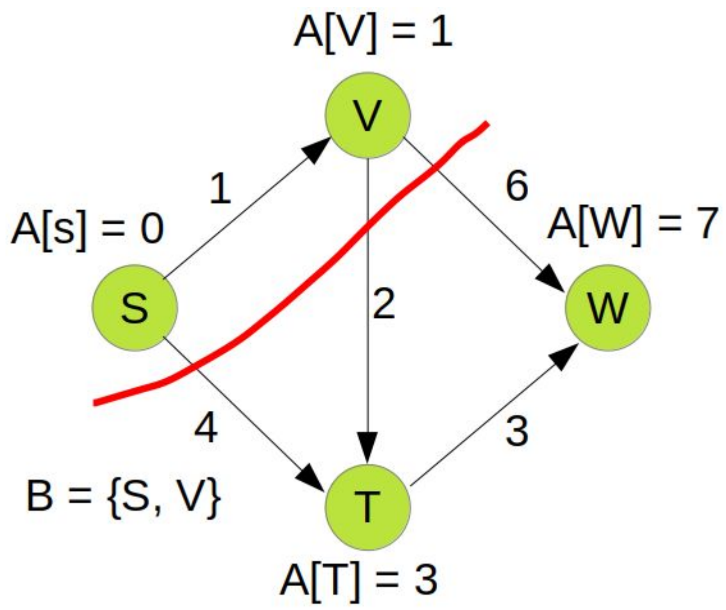


Step 2. 選擇起點 S，將 S 放入 B 形成 Cut(X, Y)，遍歷 S 所有向外的邊，更新

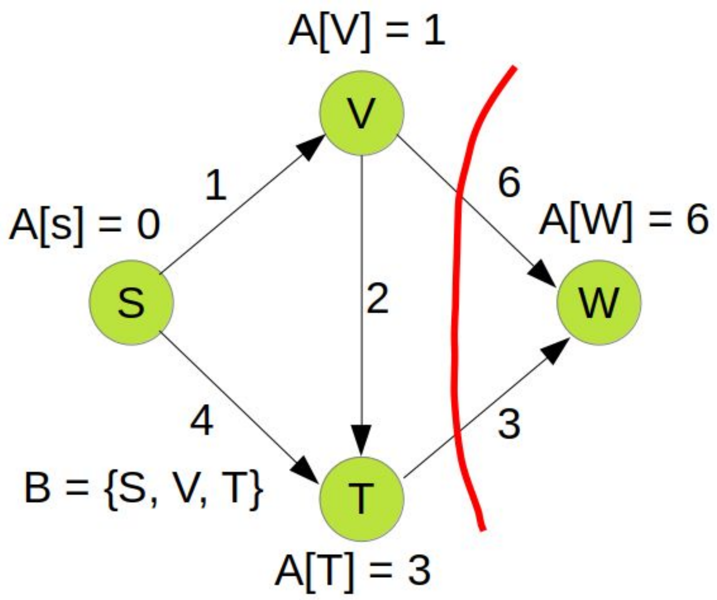
A[V] = 1 ; A[T] = 4。



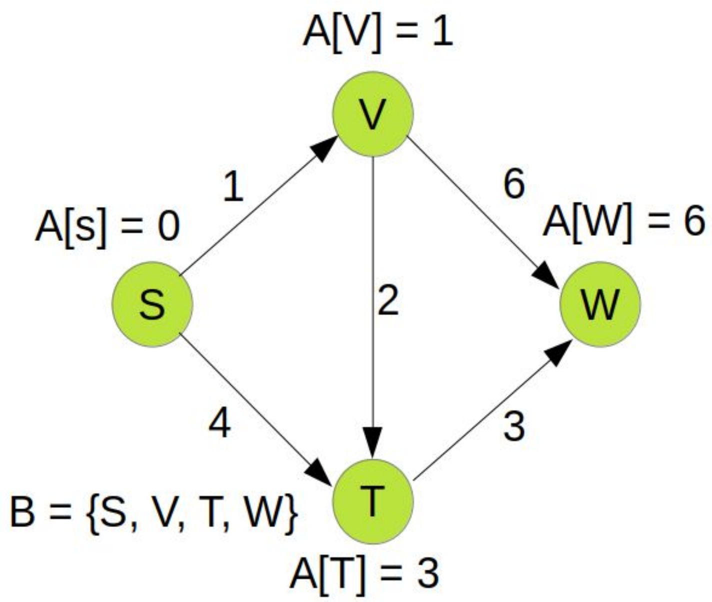
Step 3. 因 (S, V)的權重小於 (S, W)，接下來我們選擇 V ，此時由 S 到 T 有兩條路徑，一條為 S → T ; 一條為 S → V → T，第二條路徑權重為 3 ，所以 A[T] 更新為 3。同時有路徑到 W 更新 A[W] = 7。



Step 4. 接下來選擇 T [(V, T) < (V, W)]，路徑 S → V → T → W 比 S → V → W 小，更新 A[W] = 6。



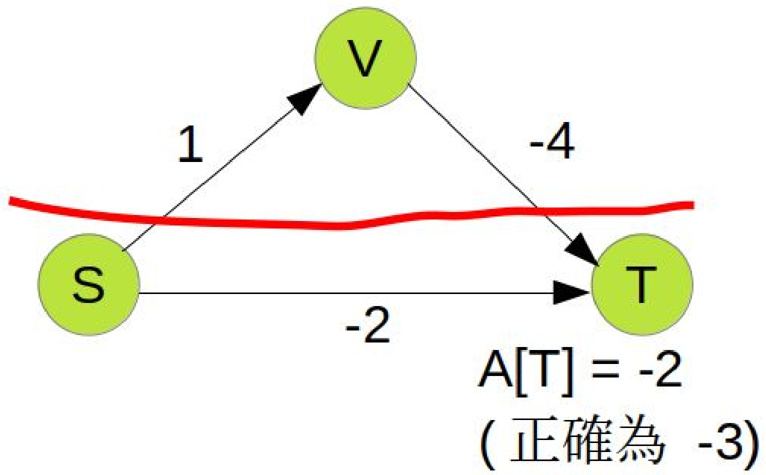
Step 5. 最後選擇 W，因 W 沒有向外的邊，且遍歷完所有點，完成算法。



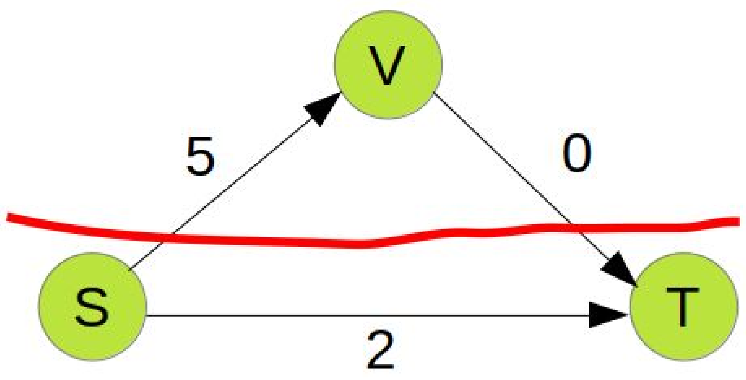
為何 Dijkstra 僅適用於權重為正的情況？

這個問題可朝兩個方向理解：

1. 考慮以下有向圖，因 (S, T) < (S, V)，因此我們會先選擇 T，這代表 A[T] = -2 在接下來的計算中，不會更新，但真正的最短路徑為 -3。

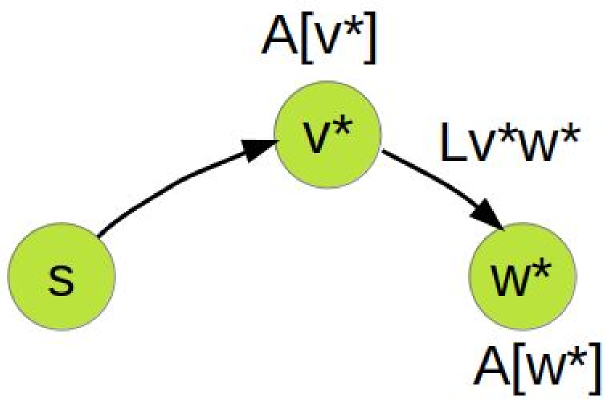


2. 有些人或許很快想到，偏移一個權重，讓所有邊都為正不就行了？我們將上圖加上 4，此時最段路徑從 S → W → T 變成 S → T，這是因為路徑權重的變化， 跟路徑經過的邊數有關，以本例來說，原本最短路徑有兩條邊，總權重變化為 8；另一條到 T 的路徑只有一條邊， 總權重變化為 4。路徑權重變化不一，導致這個想法不可行。



Dijkstra 算法的正確性

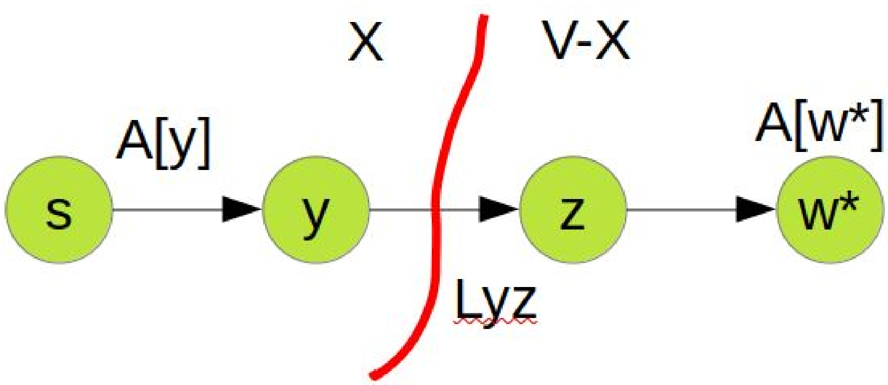
假設某個迭代之前的計算都是正確的，也就是說對於所有已探索的點 $v \in X$，$A[v] = L(v)$ ($L(v)$ 代表 起點至 v 的最短路徑)。在接下來的迭代中，我們選擇邊 (v^*, w^*)，其中 $v^* \in X$; $w^* \notin X$，$A[w^*] = A[v^*] + Lv^*w^*$，若能夠證明所有從起點到 w^* 的路徑，權重都大於 $A[w^*] = A[v^*] + Lv^*w^*$，代表 $s \rightarrow v^* \rightarrow w^*$ 一定是最短路徑。



令 P 為任意由 $s \rightarrow w^*$ 的路徑， $s \in X$; $w^* \notin X$，此任意路徑可分為三個部份：

1. X 內部的路徑，基於之前迭代都正確的假設，由 s 到 X 內節點 y，其權重應≥A[y]。
2. X 內部點 y 到外部點 z ，其權重為 Lyz。(第一次跨過 Cut)
3. 由 z 到目標點 w^* 的任意路徑，權重必定 ≥ 0。

路徑 P 的總權重至少為 $A[y] + Lyz$，依 Dijkstra 的演算策略，我們一定會從所有可能性中，選擇最短路徑 $A[v^*] + Lv^*w^* \leq A[y] + Lyz$，所以 Dijkstra 演算法一定可以找到最短路徑。



時間複雜度

最直接的 Dijkstra 演算法要遍歷所有節點(O(n))，且每個節點要遍歷所有邊(O(m))，的時間複雜度為 **O(mn)**，n 為節點數，m 為邊數，若為稠密圖 dense graph，則複雜度可能來到 O(n^3)，實在稱不上快速。

我們觀察 Dijkstra 演算法的步驟，每次都要找當前 cut(X, V − X) 權重最小的邊，有一個資料結構非常適合這種需要不斷計算最小值的狀況，也就是 **Heap**。

＊ (Heap 的插入、刪除、取出最小值的時間複雜度皆為 **O(logn)**)

＊ (所以 Prim 與 Dijkstra 的概念近乎相同)

配合 Heap 時的算法步驟如下：

1. 將起點權重設為 0，並插入 Heap。其他節點權重設為無窮大。
2. 當 Heap 不為空，則從 Heap 中取出當前最小值的點，該點設為已探索。接著遍歷該節點所有向外的邊，若該點不在 X 中，且計算出更小的路徑權重，則更新節點權重值 (刪除及重新插入)。
3. 當 Heap 為空時，代表遍歷過所有點，完成計算。

觀察上述步驟，時間複雜度主要由 Heap 相關的操作所主導：

- 每次迭代 (共 n-1 次) 取出最小值 **O(mlogn)**。
- 每一條邊都需執行一次**刪除/重新插入 O(mlogn)**

因此搭配 Heap 的時間複雜度為 **O(mlogn)**

作業——計算有向圖的最短路徑

問題描述：

檔案為一有向圖，每行第一個數字代表邊的 tail，同一行中會有數個 tuple，如 (80, 920) 代表邊的 head 及權重，所以 1 80, 920 說明邊 (1, 80) 的權重為 920。

本次作業請利用 Dijkstra 演算法，計算由節點 1 出發，至節點 7,37,59,82,99,115,133, 165,188,197 的距離。

解題方法：

無特別注意事項，利用 Dijkstra 配合 Heap 即可。