Algorithms

授課老師：張景堯

Minimum Spanning Trees/Prim

Chapter.6-Weighted Graph Algorithm

Weighted Graph Algorithms

- Beyond DFS/BFS exists an alternate universe of algorithms for *edge-weighted graphs*.
- Our adjacency list representation quietly supported these graphs:

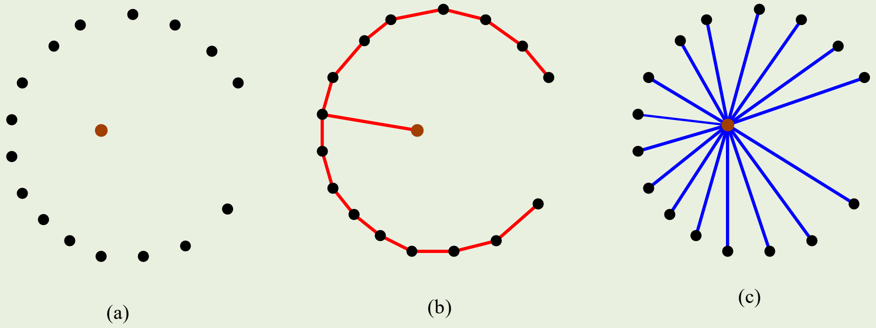
```python
class GraphEdge:
    def __init__(self, source, destination, weight):
        self.source = source
        self.destination = destination
        self.weight = weight
```

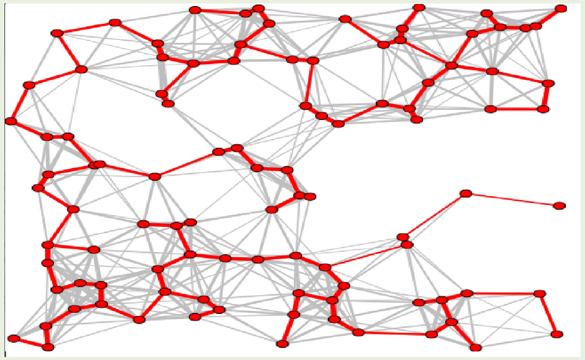
Minimum Spanning Trees (最短跨樹/最小生成樹)

- A tree is a connected graph with no cycles. A spanning tree is a subgraph of G which has the same set of vertices of G and is a tree.
- A minimum spanning tree of a weighted graph G is the spanning tree of G whose edges sum to minimum weight.
- There can be more than one minimum spanning tree in a graph → consider a graph with identical weight edges.

(在一個互連加權無向圖中一棵權值最小的生成樹，但非唯一)

Find the Minimum Spanning Tree





Why Minimum Spanning Trees?

- The minimum spanning tree problem has a long history – the first algorithm dates back to 1926!
- MST is taught in algorithm courses because:
  - It arises in many graph applications.
  - It is problem where the *greedy* algorithm always gives the optimal answer.
  - Clever data structures are necessary to make it work.
- **Greedy algorithms** make the decision of what next to do by selecting the best local option from all available choices.

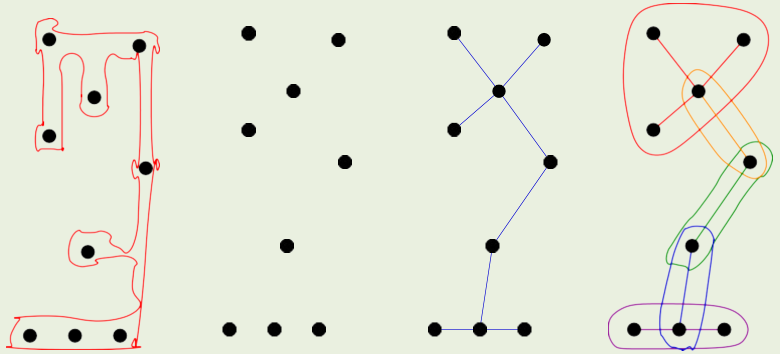
Applications of Minimum Spanning Trees

- Minimum spanning trees are useful in constructing networks, by describing the way to connect a set of sites using the smallest total amount of wire.
- Minimum spanning trees provide a reasonable way for *clustering* points in space into natural groups.
- What are natural clusters in the friendship graph?

(在一堆點上建立MST，再把幾條最大的邊刪除，即得群集)

Minimum Spanning Trees and Net Partitioning

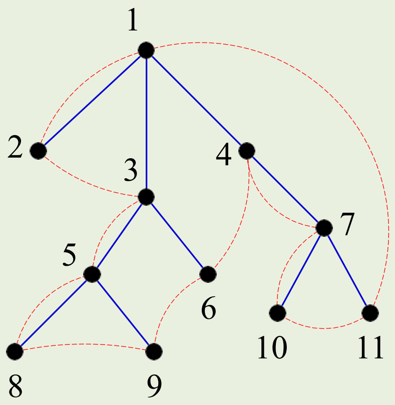
- One of the war stories in the text describes how to partition a graph into compact subgraphs by deleting large edges from the minimum spanning tree.


(a) (b) (c) (d)

(從最小生成樹中移除最大的邊就能形成2個最緊密的子圖)

Minimum Spanning Trees and TSP

- For points in the Euclidean plane, MST yield a good heuristic for the traveling salesman problem:



- The optimum traveling salesman tour is at most twice the length of the minimum spanning tree.

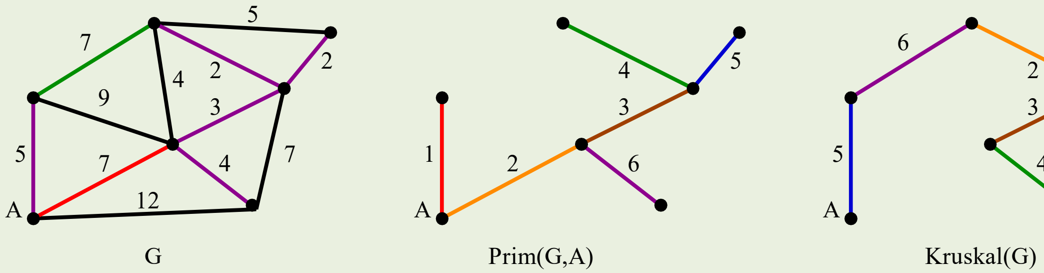
(最小生成樹的邊長總和*2可當作TSP的答案上限)

Prim’s Algorithm

- Prim’s algorithm starts from one vertex and grows the rest of the tree an edge at a time.
- As a greedy algorithm, which edge should we pick?
- The cheapest edge with which can grow the tree by one vertex without creating a cycle.

(貪婪演算法的想法就是每一步選擇都採取在目前狀態下最有利的選擇，所以我們就每一步增加一個邊納入一個頂點，就是選最小值又不會形成環狀破壞樹狀結構的邊)

Prim’s Algorithm in Action



https://upload.wikimedia.org/wikipedia/en/3/33/Prim-algorithm-animation-2.gif

Prim’s Algorithm (Pseudocode)

- During execution each vertex v is either in the tree, *fringe* (meaning there exists an edge from a tree vertex to v) or unseen (meaning v is more than one edge away). (三種頂點:在樹中、邊沿或未知)

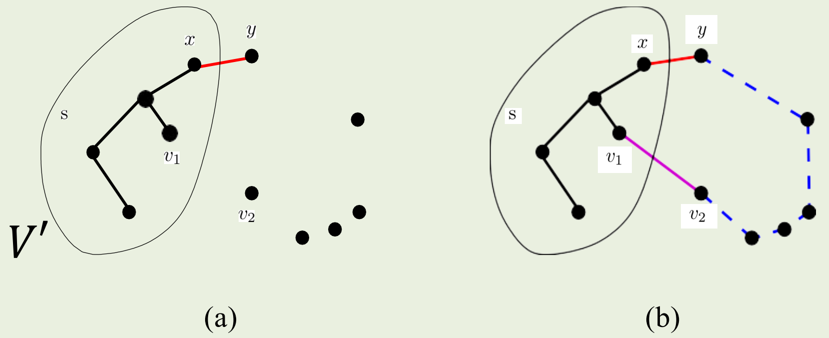
```
Prim-MST($G$)
    Select an arbitrary vertex $s$ to start the tree from.
    While (there are still non-tree vertices)
        Select min weight edge between tree/non-tree vertices
        Add the selected edge and vertex to the tree $T_{prim}$.
```

- This creates a spanning tree, since **no cycle** can be introduced, but is it minimum?

Why is Prim Correct? (Proof by Contradiction)(反證法)

- If Prim's algorithm is not correct, these must be some graph G where it does not give the minimum cost spanning tree.
- If so, there must be a first edge (x, y) Prim adds, such that the partial tree V' cannot be extended into a MST.

(假設Prim產生的不是最小生成樹，有一個邊(x, y)在當初已納入x頂點的子樹V'時就不應該把它放進最小生成樹裡)



The Contradiction (Prim)

- But if (x, y) is not in $MST(G)$, then there must be a path in $MST(G)$ from x to y, because the tree is connected.
- Let (v_1, v_2) be the other edge on this path with one end in V'.
- Replacing (v_1, v_2) with (x, y) we get a spanning tree. with smaller weight, since $\mathrm{W}(v_1, v_2) > \mathrm{W}(x, y)$. Thus you did not have the MST!!
  (Prim沒選(v_1, v_2)是因為其權值沒比(x, y)小)
- If $\mathrm{W}(v_1, v_2) = \mathrm{W}(x, y)$, then the tree is the same weight, but we couldn't have made a fatal mistake picking (x, y).
- Thus Prim's algorithm is correct!

How Fast is Prim’s Algorithm?

- That depends on what data structures are used. In the simplest implementation, we can simply mark each vertex as tree and non-tree and search always from scratch:

```
Select an arbitrary vertex to start.
While (there are non-tree vertices)
    select minimum weight edge between tree and fringe
    add the selected edge and vertex to the tree $T_{prim}$
```

- This can be done in $O(nm)$ time, by doing a DFS or BFS to loop through all edges, with a constant time test per edge, and a total of n iterations.

Prim’s Implementation(GraphVertex)

- To do it faster, we must identify fringe vertices and the minimum cost edge associated with it fast.

```python
class GraphVertex:
    def __init__(self, label):
        self.label = label
        self.intree = False #Is the vertex in the tree yet?
        self.candidate = None #Candidate mst edge
        self.parent = None #The vertex discovers me
        self.edges = []  #Edges in Adjacency List
    def __str__(self):
        output = f"{self.label}: "
        for e in self.edges:
            output += f"{self.label}-({e.weight})->{e.destination.label} "
        return output
```

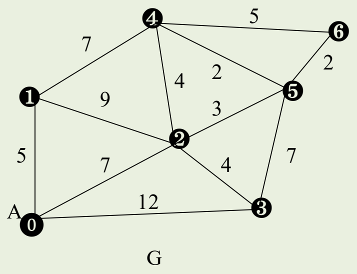
Prim's Implementation

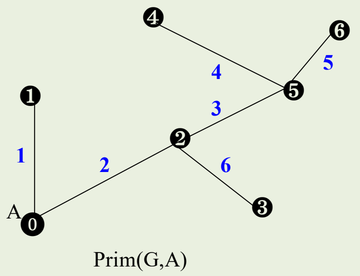
```python
def prim(graph:list)->list:
    mst = []
    for i in range(len(graph)):
        mst.append(GraphVertex(i))
    v = graph[0] #start
    while not v.intree:
        v.intree = True
        for e in v.edges:
            if not e.destination.intree:
                if e.destination.candidate is None or e.destination.candidate.weight > e.weight:
                    e.destination.candidate = e
                    e.destination.parent = v
```

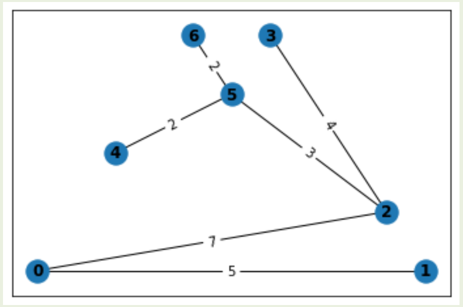
Prim’s Implementation (Cont.)

```
        dist = float('inf')
        mst_edge = None
        for i in graph:
            if not i.intree and i.candidate is not None and dist > i.candidate.weight:
                dist = i.candidate.weight
                mst_edge = i.candidate
        if mst_edge is not None: #put edge into mst graph
            s = mst_edge.source.label
            d = mst_edge.destination.label
            w = mst_edge.weight
            mst[s].edges.append(GraphEdge(mst[s],mst[d], w))
            mst[d].edges.append(GraphEdge(mst[d],mst[s], w)) #undirected graph
            v = mst_edge.destination
    return mst
```

Examples of Prim







```
mst = prim(initial_graph(7, [[0,1,5],[0,2,7],[0,3,12],[1,4,7],[1,2,9],\
        [2,3,4],[2,4,4],[2,5,3],[3,5,7],[4,5,2],[4,6,5],[5,6,2]]))
for v in mst:
    print(v)
```

Prim's Analysis

- Finding the minimum weight fringe-edge takes $O(n)$ time, because we iterate through the *distance* array to find the minimum
- After adding a vertex v to the tree, by running through its adjacency list in $O(n)$ time we check whether it provides a cheaper way to connect its neighbors to the tree. If so, update the *distance* value.
- The total time is $n \times (O(n) + O(n)) = O(n^2)$.

Minimum Spanning Trees/Kruskal

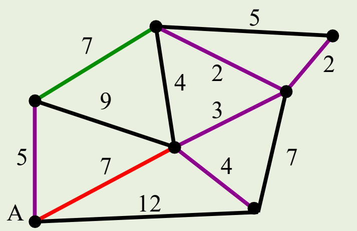
Chapter.6-Weighted Graph Algorithm

Kruskal’s Algorithm

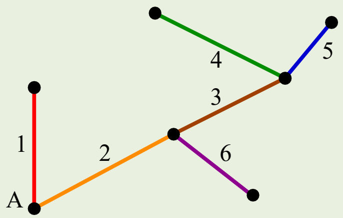
- Since an easy lower bound argument shows that every edge must be looked at to find the minimum spanning tree, and the number of edges $m = O(n^2)$, Prim’s algorithm is optimal on dense graphs.
- The complexity of Prim’s algorithm is independent of the number of edges. Kruskal’s algorithm is faster on sparse graphs.
- Kruskal’s algorithm is also greedy. It repeatedly adds the smallest edge to the spanning tree that does not create a cycle.

(Prim's時間複雜度與頂點數有關所以適用在密集圖；Kruskal's時間複雜度與邊數有關所以適用在稀疏圖)

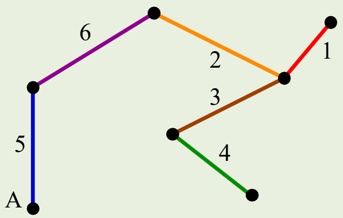
Kruskal's Algorithm in Action



G



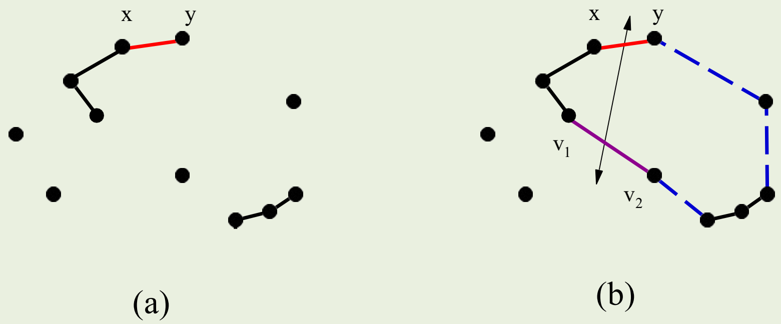
Prim(G,A)



Kruskal(G)

Kruskal is Correct (Proof by Contradiction)

- If Kruskal's algorithm is not correct, these must be some graph G where it does not give the minimum cost spanning tree.
- If so, there must be a first edge (x, y) Kruskal adds such that the set of edges cannot be extended into a minimum spanning tree.



(假設Kruskal產生的不是最小生成樹，有一個邊(x, y)不應該把它放進最小生成樹)

The Contradiction (Kruskal)

- When we added (x, y) there no path between x and y, or it would have created a cycle. Thus adding (x, y) to the optimal tree it must create a cycle.
- But at least one edge in this cycle must have been added after (x, y), so it must have heavier.
(當初能把(x, y)加進來，表示x, y之間沒有路徑可達，否則會成環；而今所宣稱的最小生成樹，若加上(x, y)就會成環狀，這表示至少有一個邊如$(\mathrm{v}_1, \mathrm{v}_2)$是在$(x, y)$後才被挑選，根據Kruskal步驟這意味著$\mathrm{w}(\mathrm{v}_1, \mathrm{v}_2) \geq \mathrm{w}(x, y)$，故發生矛盾)
- Deleting this heavy edge leaves a better MST than the optimal tree, yielding a contradiction!
- Thus Kruskal’s algorithm is correct!

How fast is Kruskal's algorithm?

- What is the simplest implementation?
  - Sort the m edges in $O(m \lg m)$ time.
  - For each edge in order, **test** whether it creates a cycle the forest we have thus far built – if so discard, else add to forest. With a BFS/DFS, this can be done in $O(n)$ time (since the tree has at most n edges).
- The total time is $O(m \lg m) + m * O(n) \rightarrow O(mn)$, but can we do better?

(接下來從改善是否成環的判斷來加速整體運算時間)

Fast Component Tests Give Fast MST

- Kruskal's algorithm builds up connected components. Any edge where both vertices are in the same connected component create a cycle. Thus if we can maintain which vertices are in which component fast, we do not have test for cycles!
(是否成環不須一個個邊去走，只需判斷要加入的邊連接的兩個頂點是否在同一個連接組件內即可)
  - *Same component*(v_1, v_2) – Do vertices v_1 and v_2 lie in the same connected component of the current graph?
  - *Merge components*(C_1, C_2) – Merge the given pair of connected components into one component.

Fast Kruskal Implementation

```
Put the edges in a min-heap
$count = 0$

while ($count < n - 1$) do
    get next edge $(v, w)$
    if (component($v$) $\neq$ component($w$))
        add to $T$
        component($v$)=component($w$) #Union two components
        $count$ = $count$ + 1
```

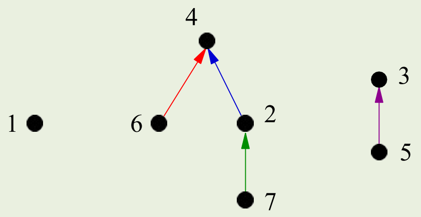
- If we can test components in $O(\log n)$, we can find the MST in $O(m \log m)$!
- Think about it: Is $O(m \log n)$ better than $O(m \log m)$?

Union-Find Programs

- We need a data structure for maintaining sets which can test if two elements are in the same and merge two sets together. These can be implemented by *union* and *find* operations, where
  - $Find(i)$ – Return the label of the **root** of tree containing element i, by walking up the parent pointers until there is no where to go.
    (以樹狀結構來維護連接組件頂點集合，回傳某樹的根頂點意味著 i 是屬於哪個連接組件)
  - $Union(i, j)$ – **Link** the root of one of the trees (say containing i) to the root of the tree containing the other (say j) so $find(i)$ now equals $find(j)$.
    (將i, j分別所屬的兩個樹狀結構結合)

Union-Find “Trees”

- We are interested in minimizing the time it takes to execute *any* sequence of unions and finds.
- A simple implementation is to represent each set as a tree, with pointers from a node to its *parent*. Each element is contained in a node, and the *name* of the set is the key at the root:



(l)

1	2	3	4	5	6	7
1	4	3	4	3	4	2

(r)

(在陣列存放頂點的parent，找最上代就是這個樹的root了)

Find Component (Recursive)

```
parents = [i for i in range(n+1)]  #initial array


def find(i):
    if i == parents[i]:
        return i
    else:
        root = find(parents[i])
        return root
```

Same Component Test

```
def same_comp(s, t):
    p1 = find(s)
    p2 = find(t)
    return p1 == p2
```

Merge Components Operation

```
def union(s, t):
    p1 = find(s)
    p2 = find(t)
    if p1 == p2:
        return True
    parents[p1] = p2
    return False
```

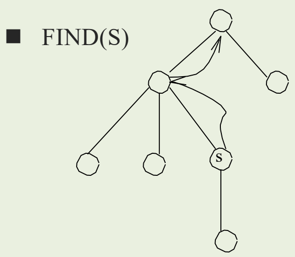
Worst Case for Union Find

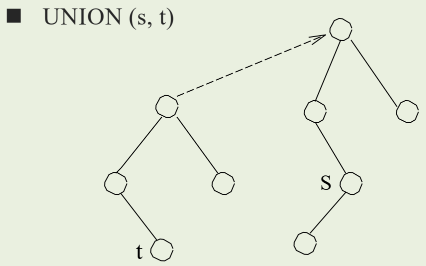
- In the worst case, these structures can be very unbalanced:

```
for i in range(1,n+1):
    union(i, i+1)
for i in range(1,n+1):
    find(1)
```

Who's The Daddy?

- We want the limit the height of our trees which are effected by *union*'s.
- When we union, we can make the tree with fewer nodes the child.





(頂點少的接到頂點較多的樹根，通常頂點少的高度也比較矮，這樣整體高度也不會增加)

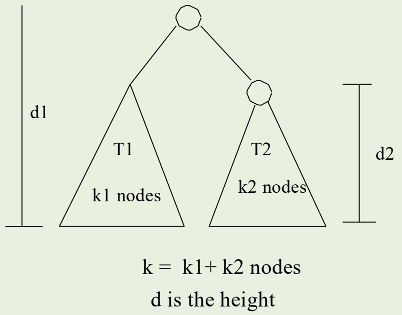
Who’s The Daddy? (Cont.)

- Since the number of nodes is related to the height, the height of the final tree will increase only if both subtrees are of equal height!
  (只有當兩個樹的頂點數/高度相同才會增加整體樹的高度)
- If $Union(t, v)$ attaches the root of v as a subtree of t iff the number of nodes in t is greater than or equal to the number in v, after any sequence of unions, any tree with n nodes has height at most $\lfloor \lg n \rfloor$.

(1個頂點的樹，假設高度是0的話，2棵樹結合，高度變1，2個高度為2的樹相結合，高度加1為2，此時頂點總數至少4個)

Proof

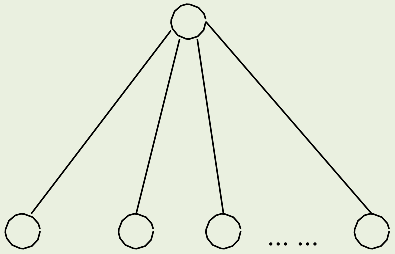
- By induction on the number of nodes k, $k = 1$ has height 0. Let d_i be the height of the tree t_i



- If $(d_1 > d_2)$ then $d = d_1 \leq \lfloor \log k_1 \rfloor \leq \lfloor \lg(k_1 + k_2) \rfloor = \lfloor \log k \rfloor$
- If $(d_1 = d_2)$ and $k_1 \geq k_2$ then
  $d = d_2 + 1 \leq \lfloor \log k_2 \rfloor + 1 = \lfloor \log 2k_2 \rfloor \leq \lfloor \log(k_1 + k_2) \rfloor = \log k$ ▮

Can we do better?

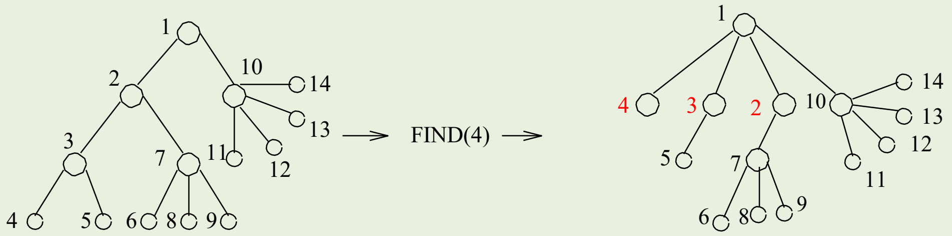
- We can do *unions* and *finds* in $O(\log n)$, good enough for Kruskal's algorithm. But can we do better?
- The ideal *Union-Find* tree has depth 1:



N-1 leaves

- On a find, if we are going down a path anyway, why not change the pointers to point to the root? (在找頂點屬於哪個樹時，就能順便調整將每個parent都直接接到根部)

Can we do better? (Cont.)



- This path compression will let us do better than $O(n \log n)$ for n union-finds.
- $O(n)$? Not quite . . . Difficult analysis shows that it takes $O(n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function and α(number of atoms in the universe)=5.

$O(n\alpha(n))$ Union Find

- Little change from original find method, other methods are the same.

```
def find(i):
    if i == parents[i]:
        return i
    else:
        parents[i] = root = find(parents[i])
        return root
```

Shortest Paths

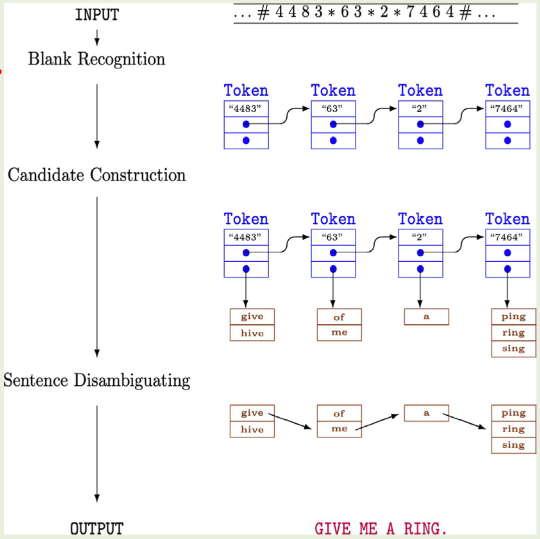
Chapter.6-Weighted Graph Algorithm

Applications for Shortest Paths

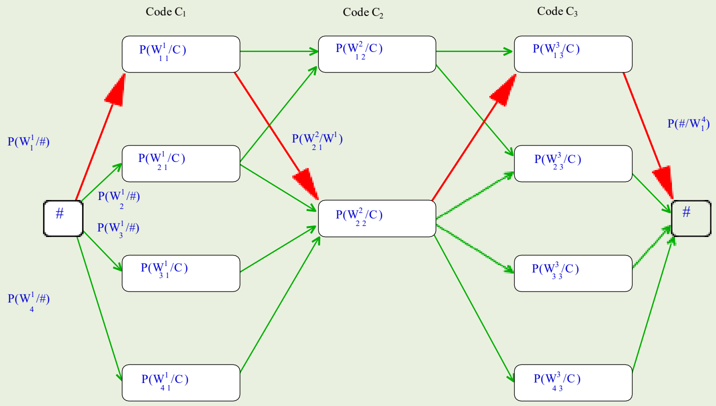
- Finding the shortest path between two nodes in a graph arises in many different applications:
  - Transportation problems(運輸) – finding the cheapest way to travel between two locations.
  - Motion planning(運動規劃) – what is the most natural way for a cartoon character to move about a simulated environment.
  - Communications problems (通訊) – how look will it take for a message to get between two places? Which two locations are furthest apart, i.e. what is the *diameter* of the network. (網路中任意兩節點間距離的最大值)

Shortest Paths and Sentence Disambiguation (最短路徑與句子歧義消除)

- In our work on reconstructing text typed on an (overloaded) telephone keypad, we had to select which of many possible interpretations was most likely.
- We constructed a graph where the vertices were the possible words/positions in the sentence, with an edge between possible neighboring words.
- The final system identified over 99% of characters correctly based on grammatical and statistical constraints.
(先建構可能單字，再跟鄰近單字組成子句，最後判定最有可能的整個句子組合)



Weighting the Graph



- The weight of each edge is a function of the probability that these two words will be next to each other in a sentence. 'hive me' would be less than 'give me', for example.
- Dynamic programming (the Viterbi algorithm) can be used to find the shortest paths in the underlying DAG.

Shortest Paths: Unweighted Graphs

- In an unweighted graph, the shortest path uses the minimum number of edges, and can be found in $O(n + m)$ time via breadth-first search.
- In a weighted graph, the weight of a path between two vertices is the sum of the weights of the edges on a path.
- BFS will not work on weighted graphs because visiting more edges can be less distance, e.g. 1 + 1 + 1 + 1 + 1 + 1 + 1 < 10. There can be an exponential number of shortest paths between two nodes – so we cannot report *all* shortest paths efficiently.

Negative Edge Weights

- Negative cost cycles render the problem of finding the shortest path meaningless, since you can always loop around the negative cost cycle more to reduce the cost of the path.
- Thus we will assume that all edge weights are ***positive***. Other algorithms deal correctly with negative cost edges.
- **Minimum spanning trees** are **unaffected** by negative cost edges.

Dijkstra’s Algorithm

- The principle behind Dijkstra’s algorithm is that if $(s,\ldots,x,\ldots,t)$ is the shortest path from s to t, then $(s,\ldots,x)$ had better be the shortest path from s to x.

(s 到 t 的最短路徑，中間經過任何一站如 x 的路徑也是 s 到 x 點的最短路徑)

- Dijkstra’s Algorithm is a greedy algorithm: making the locally optimal choice at each stage with the hope of finding a global optimum.
- This suggests a dynamic programming-like strategy, where we store the distance from s to all nearby nodes, and use them to find the shortest path to more distant nodes.

Initialization and Update

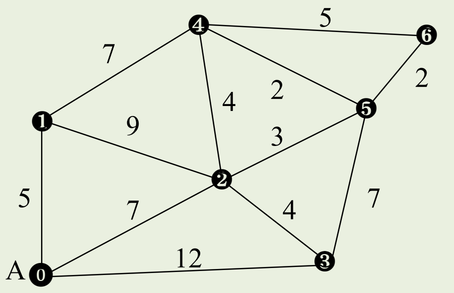
- The shortest path from s to s, $d(s,s)=0$. If all edge weights are positive, the *smallest* edge incident to s, say (s,x), defines $d(s,x)$.
- Soon as we establish the shortest path from s to a new node x, we go through each of its incident edges <u>to see if there is a **better** way from s to other nodes thru x</u>.

Pseudocode: Dijkstra's Algorithm

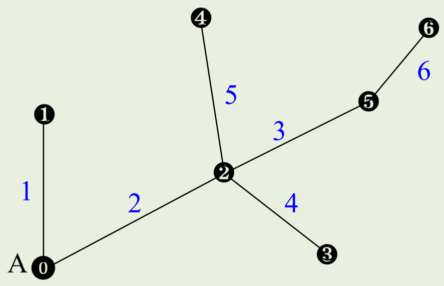
```
known = $\{s\}$
for $i$ = 1 to $n$, $dist[i] = \infty$
for each edge $(s, v)$, $dist[v] = d(s, v)$
$last = s$
while $(last \neq t)$
    select $v$ such that $dist(v) = \min_{unknown} dist(i)$
    for each $(v, x)$, $dist[x] = \boxed{\min(dist[x], dist[v] + w(v, x))}$
    $last = v$
    $known = known \cup \{v\}$
```

- This is essentially the same as Prim's algorithm.

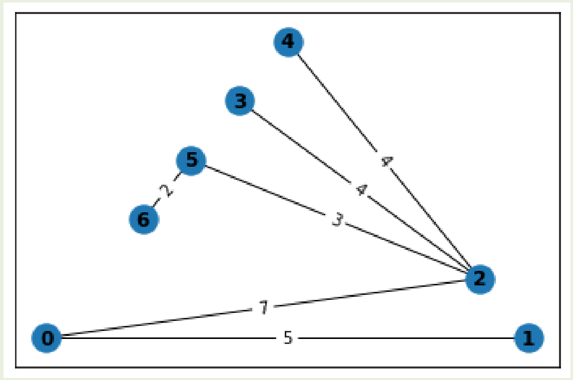
Dijkstra Example



G



Dijkstra(G,A)



Dijkstra’s Implementation(Vertex)

- See how little changes from Prim’s algorithm!

```
class GraphVertex:
    def __init__(self, label):
        self.label = label
        self.distance = float('inf') #The distance from start vertex
        self.intree = False #Is the vertex in the tree yet?
        self.candidate = None #Candidate mst edge
        self.parent = None #The vertex discovers me
        self.edges = []  #Edges in Adjacency List
    def __str__(self):
        output = "%d: " % self.label
        for e in self.edges:
            output += "%d-(%d)->%d " % (self.label, e.weight, e.destination.label)
        return output
```

Dijkstra’s Implementation

```
def dijkstra(graph:list, start:GraphVertex)->list:
    spst = []
    for i in range(len(graph)):
        spst.append(GraphVertex(i))
    v = start #start vertex
    v.distance = 0
    while not v.intree:
        v.intree = True
        for e in v.edges:
            if not e.destination.intree:
                if e.destination.distance > (e.weight + v.distance):
                    e.destination.distance = e.weight + v.distance
                    e.destination.candidate = e
                    e.destination.parent = v
```

Dijkstra’s Implementation (Cont.)

```python
        dist = float('inf')
        spst_edge = None
        for i in graph:
            if not i.intree and dist > i.distance:
                dist = i.distance
                spst_edge = i.candidate
        if spst_edge is not None: #put edge into spst graph
            s = spst_edge.source.label
            d = spst_edge.destination.label
            w = spst_edge.weight
            spst[s].edges.append(GraphEdge(spst[s],spst[d], w))
            spst[d].edges.append(GraphEdge(spst[d],spst[s], w)) #undirected graph
            v = spst_edge.destination
    return spst
```

Prim's/Dijkstra's Analysis

- Finding the minimum weight fringe-edge takes $O(n)$ time – just bump through fringe list.
- After adding new vertex v to the tree, running through its adjacency list to update the cost of adding fringe vertices if we found a cheaper way through v can be done in $O(n)$ time. The total time is $n \times n = O(n^2)$.

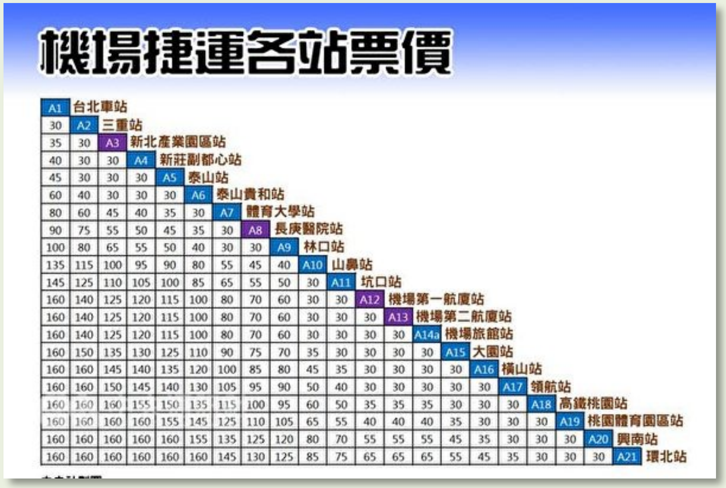
Better Data Structures = Improved Time

- An $O(m \lg n)$ implementation of Dijkstra’s algorithm would be faster for sparse graphs, and comes from using a heap of the vertices (ordered by distance), and updating the distance to each vertex (if necessary) in $O(\lg n)$ time for each edge out from freshly known vertices.
- Even better, $O(n \lg n + m)$ follows from using Fibonacci heaps, since they permit one to do a decrease-key operation in $O(1)$ amortized time.

(採用之前學過的heap或更進階的資料結構，確實可以改善Dijkstra演算法在取得最短距離步驟上的效能，但仍需考量資料大小，程式複雜度等成本效益是否值得採用)

All-Pairs Shortest Path

- Notice that finding the shortest path between a pair of vertices (s, t) in worst case requires first finding the shortest path from s to all other vertices in the graph.
- Many applications, such as finding the *center* or *diameter* of a graph, require finding the shortest path between all pairs of vertices.
- We can run Dijkstra's algorithm n times (once from each possible start vertex) to solve all-pairs shortest path problem in $O(n^3)$. Can we do better?


機場捷運各站票價

A1	台北車站																				
30	A2	三重站																			
35	30	A3	新北產業園區站																		
40	30	30	A4	新莊副都心站																	
45	30	30	30	A5	泰山站																
60	40	30	30	30	A6	泰山貴和站															
80	60	45	40	35	30	A7	體育大學站														
90	75	55	50	45	35	30	A8	長庚醫院站													
100	80	65	55	50	40	30	30	A9	林口站												
135	115	100	95	90	80	55	45	40	A10	山鼻站											
145	125	110	105	100	85	65	55	50	30	A11	坑口站										
160	140	125	120	115	100	80	70	60	30	30	A12	機場第一航廈站									
160	140	125	120	115	100	80	70	60	30	30	30	A13	機場第二航廈站								
160	140	125	120	115	100	80	70	60	30	30	30	30	A14a	機場旅館站							
160	150	135	130	125	110	90	75	70	35	30	30	30	30	A15	大園站						
160	160	145	140	135	120	100	85	80	45	35	30	30	30	30	A16	橫山站					
160	160	150	145	140	130	105	95	90	50	40	30	30	30	30	30	A17	領航站				
160	160	160	155	150	135	115	100	95	60	50	35	35	35	30	30	30	A18	高鐵桃園站			
160	160	160	160	155	145	125	110	105	65	55	40	40	40	35	30	30	30	A19	桃園體育園區站		
160	160	160	160	160	155	135	125	120	80	70	55	55	55	45	35	30	30	30	A20	興南站	
160	160	160	160	160	160	145	130	125	85	75	65	65	65	55	45	35	30	30	30	A21	環北站

Initialization

- From the **adjacency matrix**, we can construct the following matrix:

$$D[i,j] = \infty, \quad \text{if } i \neq j \text{ and } (v_i, v_j) \notin E$$
$$D[i,j] = w(i,j), \quad \text{if } (v_i, v_j) \in E$$
$$D[i,j] = 0, \quad \text{if } i = j$$

- This tells us the shortest path going through **no** intermediate nodes.

The Floyd-Warshall Algorithm

- An alternate recurrence yields a more efficient dynamic programming formulation. Number the vertices from 1 to n. Let $d[i,j]^k$ be the shortest path from i to j using only vertices from 1, 2, . . . , k as possible intermediate vertices.
- This path from i to j either goes through vertex k, or it doesn’t.

(因為是從第1頂點依序執行，經過第k頂點的路徑如果沒有比較短就不會更新，而是保留不經任何頂點或經過第1~k-1頂點中距離較短的)

- What is $d[j,j]^0$? With **no** intermediate vertices, any path consists of at most one edge, so $d[i,j]^0 = w[i,j]$.

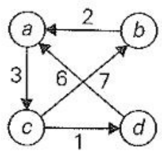
Recurrence Relation(遞迴關係)

- Adding a new vertex k helps if and only if a path goes through it, so for $1 \leq k \leq n$:

$$d[i,j]^{k} = \min(d[i,j]^{k-1}, d[i,k]^{k-1} + d[k,j]^{k-1})$$

- Computing the values of this recursive equation defines an algorithm for finding the all pairs shortest-path costs.

Floyd-Warshall Example



$$D^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \mathbf{5} & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \mathbf{9} & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e. just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$$D^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & \mathbf{9} & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e. *a* and *b* (note a new shortest path from *c* to *a*).

$$D^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \mathbf{10} & 3 & \mathbf{4} \\ b & 2 & 0 & 5 & \mathbf{6} \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \mathbf{16} & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e. *a*, *b*, and *c* (note four new shortest paths from *a* to *b*, from *a* to *d*, from *b* to *d*, and from *d* to *b*).

$$D^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & \mathbf{7} & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e. *a*, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

FIGURE 8.7 Application of Floyd's algorithm to the graph shown. Updated elements are shown in bold.

Pseudocode

- The following algorithm implements it:

```
$d^0 = w$

for $k$ = 1 to $n$
    for $i$ = 1 to $n$
        for $j$ = 1 to $n$
            $d[i,j]^k = \min(d[i,j]^{k-1}, d[i,k]^{k-1} + d[k,j]^{k-1})$
```

- This obviously runs in $\Theta(n^3)$ time, which is asymptotically no better than n calls to Dijkstra's algorithm.
- However, the loops are so tight and it is so short and simple that it runs better in practice by a constant factor.

Floyd-Warshall Implementation

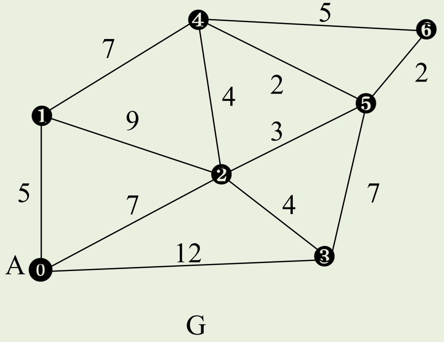
- It is notable as one of the rare graph algorithms that work better on adjacency matrices than adjacency lists.

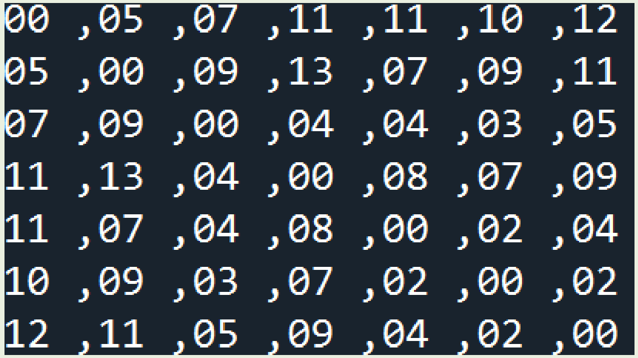
```
def floyd(M:list):
    n = len(M)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if M[i][k] != float('inf') and M[k][j] != float('inf'):
                    M[i][j] = min(M[i][j], M[i][k]+M[k][j])
```

Adjacency List to Adjacency Matrix

```
def to_adj_matrix(graph:list)->list:
    n = len(graph)
    M = [x[:] for x in [[float('inf')]*n]*n]
    for i in range(n):
        M[i][i] = 0
    for v in graph:
        i = v.label
        for e in v.edges:
            j = e.destination.label
            M[i][j] = e.weight
    return M
```

Floyd-Warshall Example



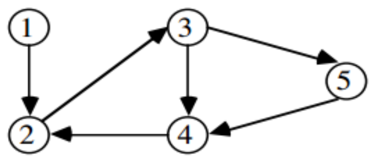


```
00 ,05 ,07 ,11 ,11 ,10 ,12
05 ,00 ,09 ,13 ,07 ,09 ,11
07 ,09 ,00 ,04 ,04 ,03 ,05
11 ,13 ,04 ,00 ,08 ,07 ,09
11 ,07 ,04 ,08 ,00 ,02 ,04
10 ,09 ,03 ,07 ,02 ,00 ,02
12 ,11 ,05 ,09 ,04 ,02 ,00
```

```
edge_list = [[0,1,5],[0,2,7],[0,3,12],[1,4,7],[1,2,9],\
        [2,3,4],[2,4,4],[2,5,3],[3,5,7],[4,5,2],[4,6,5],[5,6,2]]
graph = initial_graph(7, edge_list)
adj_M = to_adj_matrix(graph)
floyd(adj_M)
print("\n".join([",".join("%02d "%x for x in row) for row in adj_M]))
```

Transitive Closure(遞移閉包)

- The transitive closure C of a directed graph A adds edge (i, j) to C if there is a path from i to j in A.



$$A = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & & T & & & \\ 2 & & & T & & \\ 3 & & & & T & T \\ 4 & & T & & & \\ 5 & & & & T & \end{array}
\qquad
C = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & T & T & T & T & T \\ 2 & & T & T & T & T \\ 3 & & T & T & T & T \\ 4 & & T & T & T & T \\ 5 & & T & T & T & T \end{array}$$

- Transitive closure propagates the logical consequences of facts in a **database**, e.g. Is x related to y?

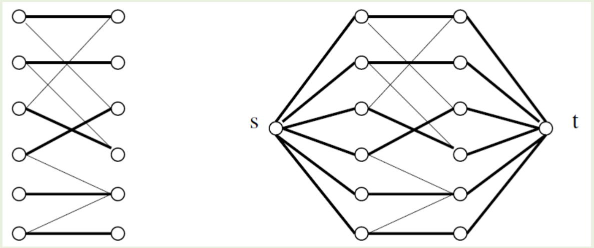
(或是判斷x到y有沒有航班到達?)

Network Flows and Bipartite Matching

Chapter.6-Weighted Graph Algorithm

Bipartite Matching(雙族匹配)

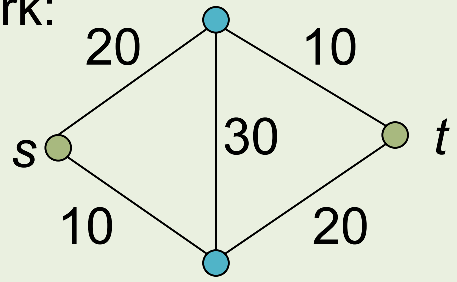
- A *matching* in a graph $G = (V, E)$ is a subset of edges $E' \subset E$ such that no two edges of E' share a vertex.
- Graph G is *bipartite* or *two-colorable* if the vertices can be divided into two sets, L and R, such that all edges in G have one vertex in L and one vertex in R.
- Matchings in these graphs have natural interpretations as job assignments or as marriages.
- The *largest bipartite matching* can be readily found using the maximum possible network flow.



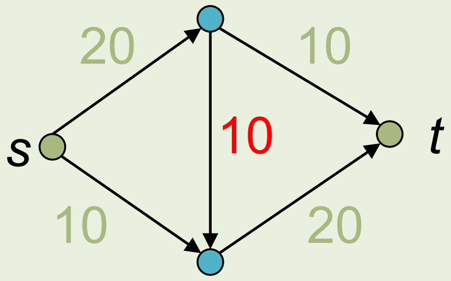
Computing Network Flows

- Traditional network flow algorithms are based on the idea of augmenting paths, and repeatedly finding a path of positive capacity from s to t and adding it to the flow.
- What is the maximum value of $f^{in}(t)$ (flow) for a given graph $G = (V, E)$? How to compute it efficiently?
- Assumption: capacities are positive integers.
- Example: $f^{in}(t) = f^{out}(s) = 30$

Network:

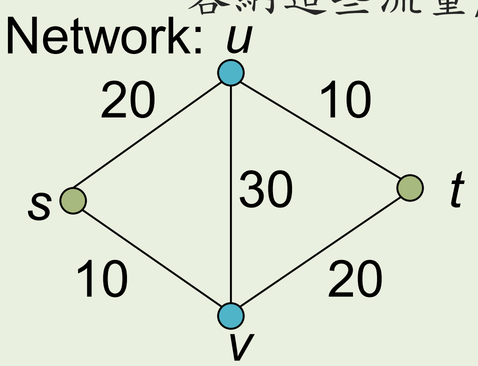


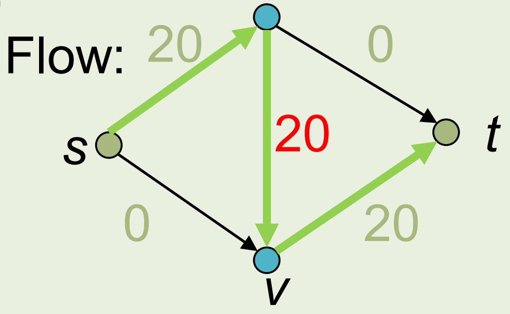
Flow:

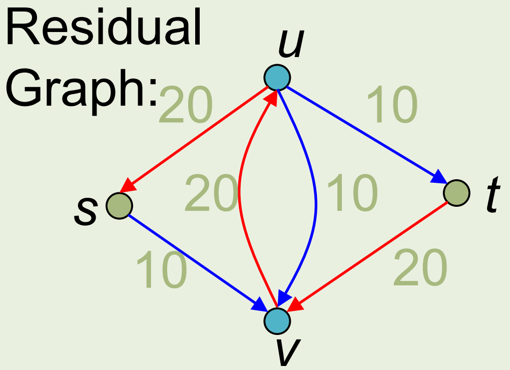


Residual Flow Graph(剩餘流量圖)

- The key structure is the *residual flow graph*, denoted as $R(G, f)$, where G is the input graph and f is the current flow through G. This directed, edge-weighted $R(G, f)$ contains the same vertices as G. For each edge (i, j) in G with capacity $c(i, j)$ and flow $f(i, j)$, $R(G, f)$ may contain two edges:
  - an edge (i, j) with weight $c(i, j) - f(i, j)$, if $c(i, j) - f(i, j) > 0$
  - an edge (j, i) with weight $f(i, j)$, if $f(i, j) > 0$. (表示這方向還能容納這些流量)



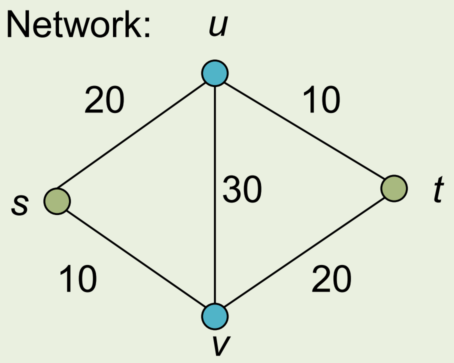


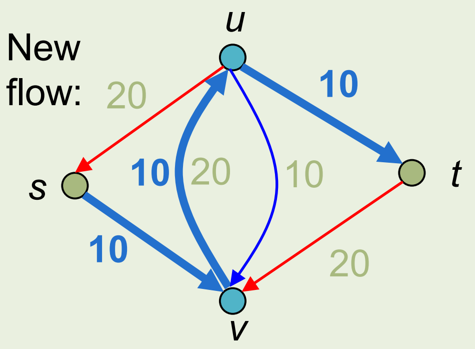


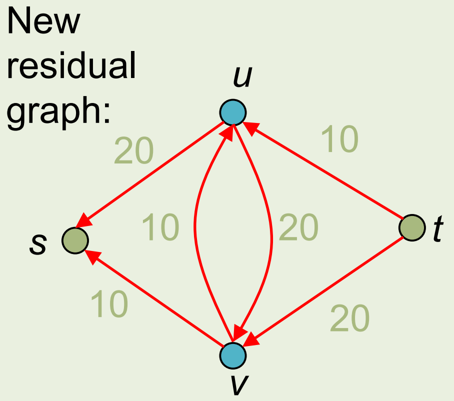
Augmenting Path(擴充路徑)

■Assume that we are given a flow f in graph G, and the corresponding residual graph $R(G, f)$

1. **Find a new flow** in residual graph: through a path with no repeating nodes, and value equal to the minimum capacity on the path (augmenting path)
2. **Update** residual graph along the path

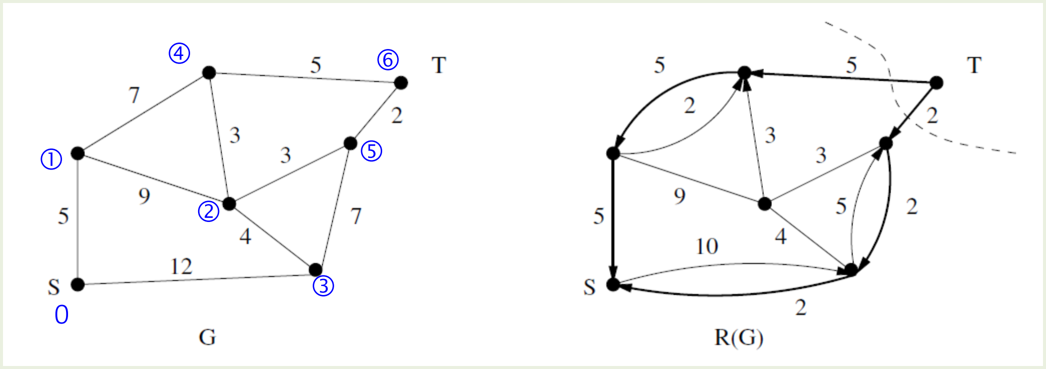






Maximum flow & minimum cut

- Maximum s–t flow in a graph G showing the associated residual graph $R(G)$ and minimum s–t cut (dotted line near t)
- A set of edges whose deletion separates s from t (like the two edges incident to t) is called an s–t cut



Design Graphs, Not Algorithms

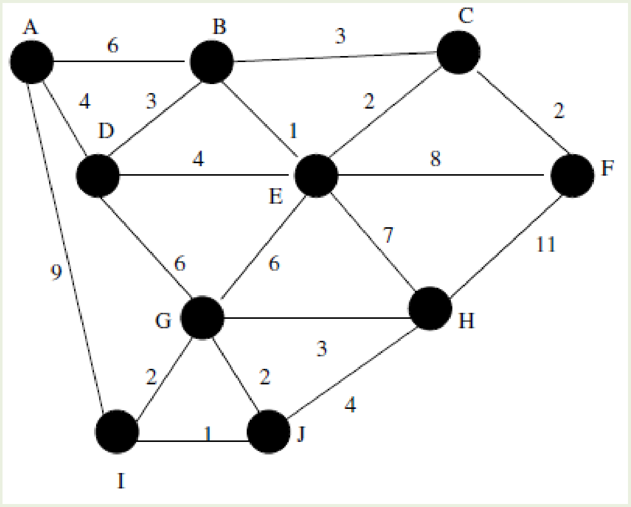
- Proper modeling is the key to making effective use of graph algorithms.
- The **maximum** spanning tree can be found by *negating the edge weights* of the input graph G and using a minimum spanning tree algorithm on the result. The most negative weight spanning tree will define the maximum weight tree in G.
- To solve bipartite matching, we constructed a special network flow graph such that the maximum flow corresponds to a maximum cardinality matching.

(將問題對應到可用現成演算法解決的圖形模型，而不是重新打造一個新演算法。)

Exercises

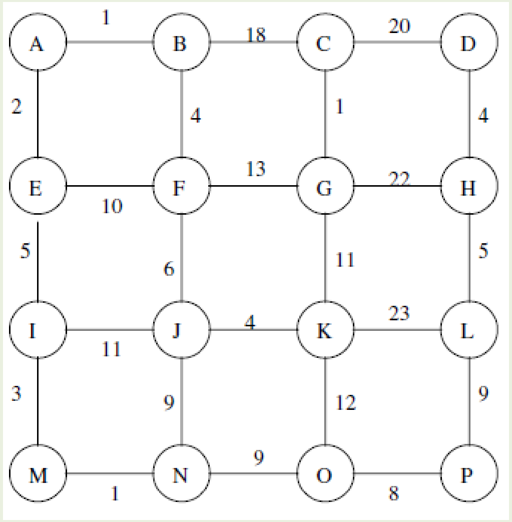
Problems of the Day

- For the following graph, write down the order of edges put into Minimum Spanning Tree in Prim's algorithm (start from vertex A). If two edges with the same weight, picking the vertices of edge in alphabetical order (A before Z). Answer looks like: (A,D),(B,D),(B,E),…



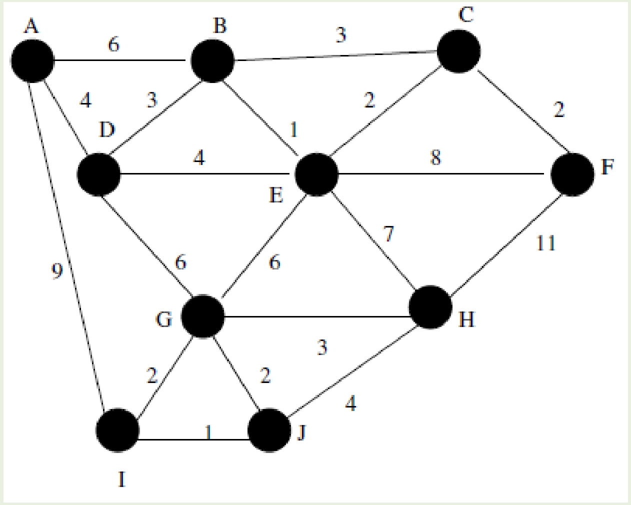
Problems of the Day (Cont.)

- For the following graph, write down the order of edges put into **Minimum Spanning Tree** in Kruskal's algorithm. If two edges with the same weight, picking the vertices of edge in alphabetical order (A before Z).
  Answer looks like: (A,B),(C,G),(M,N),(A,E),…



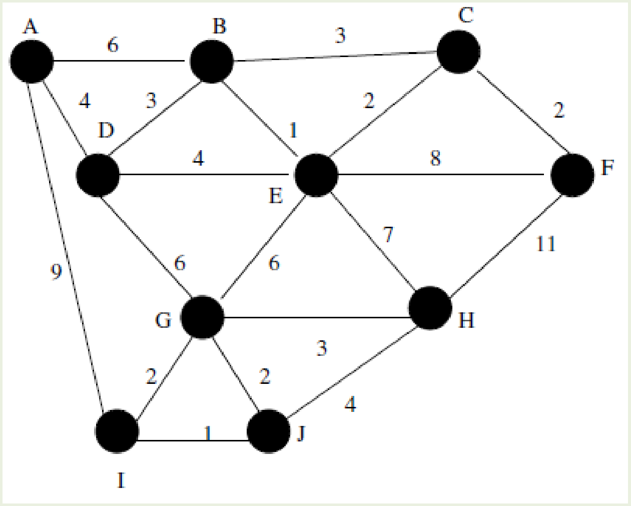
Problems of the Day (Cont.)

- For the following graph, find the **shortest path spanning tree** rooted in A and write down the order of edges. If two edges with the same weight, picking the vertices of edge in alphabetical order (A before Z). Answer looks like: (A,D),(A,B),(B,E),...



Problems of the Day (Cont.)

- For the following graph, compute the maximum flow from *A* to *H*.



Problems of the Day(Cont.)

- Let $G = (V, E)$ be an undirected weighted graph, and let T be the minimum spanning tree rooted at a vertex v. Suppose now that the edge weights in G are increased by a constant number k. Is T still the **minimum spanning tree** from v? ~~Why?~~

- Let $G = (V, E)$ be an undirected weighted graph, and let T be the shortest-path spanning tree rooted at a vertex v. Suppose now that the edge weights in G are increased by a constant number k. Is T still the **shortest-path spanning tree** from v? ~~Why?~~

Program Exercises (Moodle CodeRunner)

- Exercise 07 (close at 4/29 23:59)
  - Traffic Flow :
    - *A city has n intersections and m bidirectional roads connecting pairs of intersections. To take as input a city map representing as an adjacency list which is a list of instances of Intersection class, return the value of the minimum capacity among remaining roads.*
    - *Hint: Get the Maximum Spanning Tree, then find the road with minimal capacity in tree.*
  - Road Network :
    - *The Department of Transportation is considering adding a new section of highway to the Highway System. Each highway section connects two cities. City officials have* submitted *proposals for the new highway. Determine the best proposed highway section which leads to the most improvement in the total driving distance.*