No:
Date: /

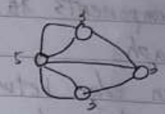
Graph Basics

Seven Bridges of Königsberg

.只要兩 class { 7 個橋 (線)
4 個陸地 (点)

$G = \{V, E\}$

$V(G)$: vertex set (点)



$E(G)$: edge set (邊)

degree : number of edges

$\sum degree(V_i) = |E(G)| \times 2$ 恆成立

Eulerian path (trial) / Euler walk

- vists every edge exactly once
  - · 0 or 2 nodes with odd degrees (剩下都不可能)
    (1 不會發生, odd degreess 不可能奇數個) 公式

Eulerian circuit (cycle) / Euler tour

- begin and end at the same vertex
  - · 0 node with odd degrees

Basic Terminologies

· Undirected graph (無向) (双向)

· Directed graph (digraph) (有向) (單向)

· Adjacent vertices (相鄰的) 有一個邊相連

· Edge is incident to vertices (有关的)

· Path : a sequence of edges (路徑) (length 距離)
small world — 找6 次朋友的朋友可找到老朋友

· Cycle : begin & end at the same vertex

· Simple path : a path that passes through any vertex only once (每个点只走一次)

· Simple cycle : a cycle that passes through the other vertices only once
(起点終点一樣且每个点只走一次)

No:
Date: /

Connected graph
- There is a path between any two vertices
Disconnected graph
connected components 相連的一部分 (subgraphs)
Complete graph
- There is an edge between any two verties (任兩点皆相連)
Strong connected graph
- For any two verties on a digraph, there is a path from one vertex to the other (有向圖 所有点可互相到達)
Weighted graph
- the edges have numeric labels (邊上有數字集合)(權重)

Graphs as ADTs
Variations of an ADT graph are possible
- Vertices may or may not contain values
  · Many problems have no need for vertex values
  · Relationships among vertices is what is important
- Either directed or undirected edges
- Either weighted or unweighted edges
Insertion and deletion operations for graphs apply to vertices and edges
Graphs can have traversal operations

No:
Date: /

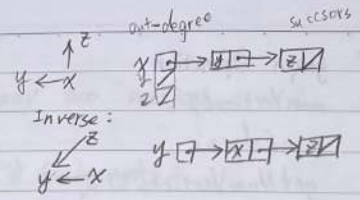
ADT graph Operation
```
int      numVertices;
int      numEdges;
int      getNumVertices();
int      getNumEdges();
int      getWeight(Edge e);
void     add(Edge e);
void     remove(Edge e);
bool     isEdge(Vertex u, Vertex v);
int      getDegree(Vertex v);
bool     isConnected(Graph g);
edgeList traverse(Graph g);
```

Graph Representations
Most common implementations
1. Adjacency matrix
2. Adjacency list
Adjacency matrix for a graph that has n vertices numbered $0, 1, \dots, n-1$
- An n by n array matrix such that matrix [i][j] indicates whether an edge exists from vertex i to vertex j

Adjacency Matrix traverse (g): $O(|V|^2)$
For an unweighted graph, matrix [i][j] is
· 1 (or true) if an edge exists from vertex i to vertex j
· 0 (or false) if no edge exists from vertex i to vertex j
For a weighted graph, matrix [i][j] is
· The weight of the edge from vertex i to vertex j
· ∞ (or 0) if no edge exists from vertex i to vertex j

No:
Date: /

Adjacency list



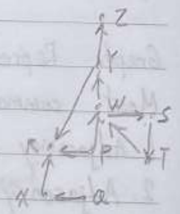
traverse(g) : O(|V| + |E|)
(if it is a sparse matrix)

Graph Representations

Two common operations on graphs
1. Determine whether is an edge from vertex i to vertex j   isEdge(i,j)
2. Find all vertices adjacent to a given vertex i   getDegree(i)

- Adjacency matrix
  - Supports operation 1. more efficiently
- Adjacency list
  - Supports operation 2. more efficiently
  - Often requires less space than an adjacency matrix

Sequential representation



- nodes + edges

把点編號　P Q R S T W X Y Z / 0 1 2 3 4 5 6 7 8　　9以後是邊

[0] [1] [2] [3] ... [7] [8] [9] [10] [11] [12] [13] ...
10 12 13 14 19 20 20 2 5 6 6

12−10=2
表示 [0] 有 2 個邊　　Q → X 1 → 2 像橋梁
用來相互或相減

圖更改較麻煩

undirected graph : |V| + 2|E| + 1 (無向兩倍)

No:
Date: /

Graph Traversals

每個点都走一次 (Visits all the vertices that it can reach)
Visits all vertices of the graph if and only if the graph is connected
- A connected component
  - The subset of vertices visited during a traversal that begins at a given vertex

To prevent indefinite loops (break the cycles)
- Mark each vertex during a visit, and never visit a vertex more than once

DFS and BFS Traversals

Depth-First Search (DFS) Traversal 深度優先
-last visited, first explored (堆疊)

Breadth-First Search (BFS) Traversal
- first visited, first explored (Queue)

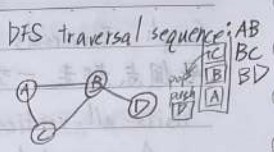
DFS

-Has a simple recursive form
可用在 有向、無向
Has an iterative form that uses stack

```
recursiveDFS(Vertex v)
    Mark v as visited;
    for (each unvisited vertex u adjacent to v)
        recursiveDFS(u);
```

No:
Date: /

DFS traversal sequence: AB BC BD

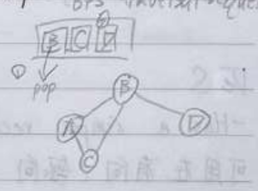


```
iterative DFS ( Vertex v)
  s.createStack ();
  s.push(v);
  Mark v as visited;
  while (!s.isEmpty()){        if (any visited vertex adjacent to u)
    u = s.getTop();              Cycle is found!
    if (unvisited vertex w is adjacent to u){
      s.push(w);
      Mark w as visited; //output
    }
    else s.pop();
}
```

BFS

- An iterative form uses a queue
- A recursive form is possible, but not simple

~~BFS traversal sequence:~~ AB AC BD



```
iterative BFS ( Vertex v)
  q.createQueue();
  q.enqueue(v);
  Mark v as visited;
  while(!q.isEmpty()){
    q.dequeue(u);          if (any visited vertex adjacent to u)
                             Cycle is found!
    for( each unvisited vertex w adjacent to u){
      Mark w as visited; //output
      q.enqueue (w);
    }
}
```

會形成 spanning trees

No:
Date: /

```
q.createQueue();
Mark v as visited;
recursive BFS (v);

recursive BFS ( Vertex v)
  for (each unvisited vertex u adjacent to v){
    Mark v as visited;
    q.enqueue(u);
  }
  while(!q.isEmpty()){
    q.dequeue (w);
    recursive BFS(w);
  }
```

No:
Date: /

No:
Date: /

Graph Applications

Topological Sort 拓撲排序

Topological order

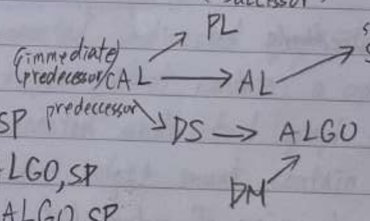
- 有向圖且無 cycles (Acyclic Digraph or Directed Acyclic Graph) DAG
- Several topological orders are possible for a given graph

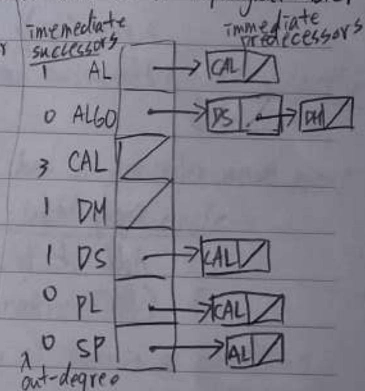
Topological sorting

- Arranging the vertices into a topological order

topSort1

1. Find a vertex that has no successor (out-degree = 0)
2. Add the vertex to the begining of a list
3. Remove that vertex from the graph, as well as all edges that lead to it
4. Repeat the previous steps until the graph is empty
   - When the loop ends, the list of vertices will be in topological order





- SP
- ALGO, SP
- DM, ALGO, SP
- PL, DM, ALGO, SP
- DS, DM, ALGO, SP
- AL, DS, DM, ALGO, SP
- CAL, AL, DS, DM, ALGO, SP

- ALGO (DM, DS = 0)
- DM, ALGO
- DS, DM, ALGO (CAL = 2)
- PL, DS, DM, ALGO (CAL = 1)
- SP, PL, DS, DM, ALGO (AL = 0)
- AL, SP, PL, DS, DM, ALGO (AL = 0)
- CAL, AL, SP, PL, DS, DM, ALGO

* indegree = 0 (no predecessor)

No:
Date: /

topSort2
- A modification of the iterative DFS algorithm
- Push all vertices that have no predecessor onto a stack
- Each time you pop a vertex from the stack, add it to the begining of a list of vertices
- When the traversal ends, the list of vertices will be in topological order

```
iterativeDFS(Vertex v)
    s.createStack();
    s.push(v);
    Mark v as visited;
    while(!s.isEmpty()){
        u=s.getTop();
        if(unvisited vertex w is adjacent to u){
            s.push(w);
            Mark w as visited;
        }
        else{
            s.pop();
            Add it to the beginning of output list.
        }
}
```

No:
Date: /

Spanning Tree 生成樹
- undirected connected graph without cycles (acyclic)
- A subgraph of G that contains all of G's vertices and enough of its edges to form a tree

ex: CISCO, Spanning Tree Protocol (STP) 網路(通訊協定)
(加邊)connected ↔ acyclic (把一些邊拿掉)
(Spanning Tree要剛好)

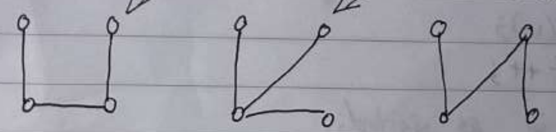
To obtain a spanning tree from a connected undirected graph with cycles
- Remove edges until there are no cycles

Detecting a cycle in an undirected connected graph
- A connected undirected graph that has n vertices must have at least n-1 edges
- A connected undirected graph that has n vertices and exactly n-1 edges cannot contain a cycle
- A connected undirected graph that has n vertices and more than n-1 edges must contain at least one cycle

4個點三個邊有2個構造 (isomorphic 同構)
Two graphs are isomorphic if and only if there is a bijection f between their vertex sets

1. {2, 2, 1, 1} degree
2. {3, 1, 1, 1}

P(4,2)
(C(4,3))
↓
12+4=16
有16種畫法



6-4=2

No:
Date: /

Various vertex labeling: n^{n-2}

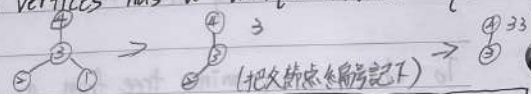
2 nodes → $2^{2-2} = 1$

3 nodes → $3^{3-2} = 3$

4 nodes → $4^{4-2} = 16$

Prüfer sequence 普吕弗序列

1. Each labeled tree with n vertices has a unique Prüfer sequence of length n-2
   - Conversion algorithms
     - Leaf with the smallest label
     - Keep the label of its parent



建立 degree array: [1 2 3 4]
　　　　　　　　　　1 1 3 1

2. Each Prüfer sequence of length n-2 has a unique labeled tree with n vertices

   33 → 表 3 是内部節点 → degree = 1+2 = 3

DFS / BFS for Spanning Trees

```
iterativeDFS(Vertex v)
    s.createStack();
    count = 0;
    s.push(v);
    Mark v as visited
    while(!s.isEmpty() && count < |v|-1){
        u = s.getTop();
        if(unvisited vertex w is adjacent to u){
            s.push(w);
            count ++;
            Mark w as visited;
        }
        else s.pop();
}
```

No:
Date: /

```
iterativeBFS(Vertex v)
    q.createQueue();
    count = 0;
    q.enqueue(v);
    Mark v as visited;
    while(!q.isEmpty() && count < |v|-1){
        q.dequeue(u);
        for(each unvisited vertex w adjacent to u){
            Mark w as visited;
            q.enqueue(w);
            count ++;
        }
    }
```

Minimum Spanning Tree 最小生成樹

Cost of spanning tree
- Sum of the edge weights on a spanning tree

A minimum spanning tree of a connected undirected graph has a minimal edge-weight sum
- A particular graph could have several minimum spanning trees

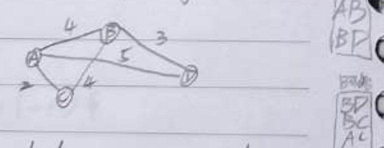
Other variations
- (minimum) Steiner tree (觀念一樣)
- K-minimum spanning tree

Prim's Algorithm

Find a minimum spanning tree that begins at any given vertex

1. Find the least-cost edge(v,u) from a visited vertex v to some unvisited vertex u
2. Mark u as visited
3. Add the vertex u and the edge (v,u) to the minimum spanning tree
4. Repeat the above steps until all vertices are visited

Minimum Spanning Tree (MST)

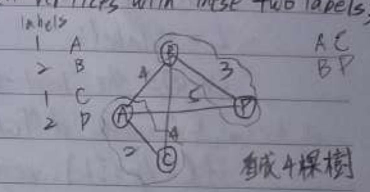



```
PrimAlgorithm(Vertex v)
  Mark v as visited;
  count = 0;
  while( count < |V|-1 )
    (v,u)= the least-cost edge from visited to unvisited (Priority Queue)
    Mark u as visited;
    Add (v,u) into MST;
    count ++;
```

Kruskal's Algorithm

1. Create a forest, where each vertex is a tree
2. Find the least-cost edge(v,u) where vertex v and vertex u are from two different trees
3. Merge the trees of vertex v and vertex u, and add the edge(v,u) to the minimum spanning tree
4. Repeat the above steps until |V|-1 edges

```
KruskalAlgorithm()
  Assign a unique label to each vertex;
  count = 0;
  while( count < |V|-1 )
    (u,v)= the least-cost edge of two vertices with different labels
    Assign the label min(u,v) to all vertices with these two labels;
    Add (v,u) into MST;
    count ++;
```



剩成4棵樹

No:
Date: /

Sollin's Algorithm

1. Create a forest, where each vertex is a tree
2. For each tree T, do the following steps:
   1) Find the least-cost edge(v, u) where vertex v is in T and vertex u is outside T
   2) Merge the trees of vertex v and vertex u, and add the edge(v, u) to the minimum spanning tree
3. Repeat step 2 until only one tree is left

```
SollinAlgorithm()
Assign a unique label to each vertex: size = |v|;
while(size > 1)
  Initialize Edges[1...size] as empty sets;
  for each vertex v
      L = v.label;
      (v, u) = the least-cost edge from v to u for any vertex with
               a different label;
      if(Edges[L].weight > (v, u).weight)
          Edges[L] = (v, u);
  for each edge(v, u) in Edges but not in MST
     Assign min(v.label, u.label) to vertices in the sets of
     v and u;
     Add (v, u) to MST;
     size--;
```

No:
Date: /

Shortest Path

路徑 - 由很多邊組成

Shortest path between two vertices in a weighted graph is the path that has the smallest sum of its edge weights

Dijkstra's Algorithm

- Find the shortes paths between a given origin and all other vertices

Basic idea

- A set vertexSet of selected vertices
- An array weight, where weight[v] is the cheapest weight of the shortest path from vertex 0 (origin) to vertex v that passes through only the vertices in vertexSet

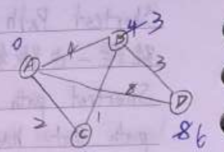
n-1步

1. Initialize vertexSet & weight; $v = v_0$;
2. Update weight for each vertex u not in vertexSet, which is adjacent to v
   weight[u] = min{ weight[u], weight[v] + edgeWeight[v,u] }
3. Find the shortest path from 0 to u among every path that starts from 0, passes vertices in vertexSet, and ends at a vertex not in vertexSet
   if(weight[u] is minimum) vertexSet = vertexSet + {u}
4. Repeat steps 2,3 until no more vertex can be added

No:
Date: /

Dijkstra Algorithm(Vertex v_0)

```
weight[0…n] = {0, ∞, …, ∞};
vertexSet = ∅;
do{
    Add v into vertexSet;
    for edge(v, u) where u is not in vertexSet
        weight[u] = min{ weight[u], weight[v]+edgeWeight[v,u]}
    cheapest = ∞;
    for vertex u not in vertexSet
        if( weight[u] < cheapest ){
            v = u;
            cheapest = weight[u];
        }
} while ( cheapest < ∞ );
```



找 A → B [dB] > [c] + edgew[cB]

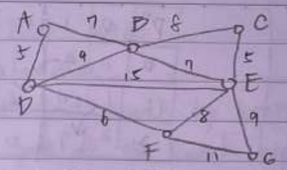
	A	B	C	D
A	0	4	2	8
B	4	0	1	3
C	2	1	0	∞
D	8	3	∞	0

A→B : 4
A→C→B : 2+1=3

若最小合成樹唯一，會和 Dijkstra 答案不一樣
↑ sum　　　　　↑ 路線

Kruscal、Prime 答案一樣(順序不一樣)

No:
Date: /



{vertexSet = {A}
weight = {0, 7, ∞, 5, ∞, ∞, ∞}

{vertexSet = {A, D}
weight = {0, 7, ∞, 5, 20, 11, ∞}

{vertexSet = {A, D, B}
weight = {0, 7, 15, 5, 14, 11, ∞}　　耗時間不耗空間

{vertexSet = {A, D, B, F, E, C}
weight = {0, 7, 15, 5, 14, 11, 22}

All Pairs Shortest Paths

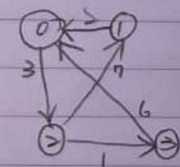
任何起点到目地的最短距離

Floyd's Algorithm

1. Initialize distance matrix D^{-1} = adjacency matrix ;
2. For k = 0 to |v|-1
   $D^k \leftarrow D^{k-1}$; // Add vertex k into vertexSet
   For i = 0 to |v|-1
   For j = 0 to |v|-1
   終 $D[i,j] = \min\{ D^{k-1}[i,j],\ D^{k-1}[i,k] + D^{k-1}[k,j] \}$;

起

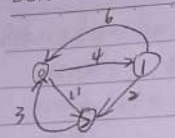
D^{-1}	0	1	2	3
0	0	∞	3	∞
1	2	0	∞	∞
2	∞	7	0	1
3	6	∞	∞	0



D^0	0	1	2	3
0	0	∞	3	∞
1	2	0	5	∞
2	∞	7	0	1
3	6	∞	∞	0

D^{-1}: all-pairs shortest paths with no intermediate vertex
D^0: all-pairs shortest paths with intermediate vertex 0

No:
Date: /



D^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

D^{0}	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

D^{1}	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

D^{2}	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

Summary

- Topological sorting produces a linenear order of the vertices in a directed graph without cycles
- Trees are connected undirected graphs without cycles
- A spanning tree of a connected undirected graph is
  - A subgraph that contains all the graph's vertices and enough of its edges to form a tree
- A minmum spanning tree for a weighted undirected graph is
  - A spanning tree whose edge-weight sum is minimal
- The shortest path between two vertices in a weighted directed graph is
  - The path that has the smallest sum of its edge weights

No:
Date: /

Best-First Search

A^* algorithm

- Best-first search by keeping a priority queue and traversing a path of the lowest expected total cost
- Combines two pieces of information
  - Dijkstra's algorithm: favo vertices close to the origin
  - Greedy best-first search: favor vertices close to the goal
    - ↳先解小問題選最好的去解大問題
- Expected total cost: $f(v) = g(v) + h(v)$
  - $g(v)$: exact cost of the path from the origin to vertex v
  - $h(v)$: heuristic estimated cost from vertex v to the goal
    - It helps efficiency if $h(v)$ is a lower bound of actual cost

No:
Date: /

Graph Problems

Activity-on-Vertex (AOV) Network

Activity-on-Edge (AOE) Network

Directed edge: activity (task) to be performed
Vertex: event to signal the completion of certain activities
Edge weight: the time required to perform an activity
Path length: the total time from the start to the last event
Critical Path: a path with the longest length
- the minimum time required to complete the project

· 不能有 cycle

Critical Path Analysis

Input
- A list of all activities required to complete the project
- The time (duration) that each activity will take to completion
- The dependencies between the activities

Output
- The longest path of planned activities to the end of the project
- The earliest time and the latest time that each activity can start and finish without making the project longer
- Determines "critical" activities (on the longest path)
- Prioritize activities for the effective management and to shorten the critical path of a project

Determine Critical Paths
- Delete all non-critical activities (nonzero slack)
- Generate all the paths from the start to the end

Speed up the activities on all critical paths
- resource can be concentrated on these activities in an attempt to reduce the project completion time (所有路徑都會經過的)

No: Date: /

Critical Path Method : Forward Phase

Like topSort!

1. Find vertex that has no successor (out-degree = 0)
2. Add v to the beginning of a list
3. Remove v from the graph, as well as all edges that lead to v
4. Repeat the previous steps until the graph is empty

⇓

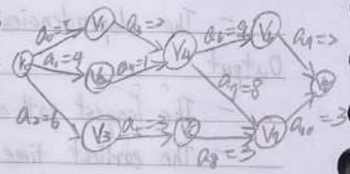
1. Find vertex v that has no predecessor (in-degree = 0)
2. For each immediate successor u, do the following:
   - Set ea [x] = ee[v], where x is the activity on <v,u>
   - Set ee [u] = max{ ee [u] ee[v] + duration of <v,u>}
   - Decrease the in-degree of u
3. Repeat the steps until all vertices are visited
   - For the vertex w that has no successor, le[w] = ee[w] !

in-degree activity duration

V_0 [0] → [a_0 | 5] → [a_1 | 4] → [a_3 | 6]
V_1 [1] → [a_3 | 2]
V_2 [1] → [a_4 | 1]
V_3 [1] → [a_5 | 3]
V_4 [2] → [a_6 | 9] → [a_7 | 8]
V_5 [1] → [a_8 | 3]
V_6 [1] → [a_9 | 2]
V_7 [2] → [a_{10} | 3]
V_8 [2]

queue: V_1, V_2, V_3

a_0	V_0	V_1
a_1	V_0	V_2
a_2	V_0	V_3
a_3	V_1	V_4
a_4	V_2	V_4
a_5	V_3	V_5
a_6	V_4	V_6
a_7	V_4	V_7
a_8	V_5	V_7
a_9	V_6	V_8
a_{10}	V_7	V_8



ea	ee
[0]:0	[0]: 0
[1]:0	[1]: 5
[2]:0	[2]: 4
[3]: 5	[3]: 6

V_3最早開始時間

No: Date: /

Backward

1. Find vertex u that has no successor (out-degree = 0)
2. For each immediate predecessor V, do the following:
   - Set la [x] = le[u] - duration of <v,u>, where x is the activity on <v,u>
   - Set le[v] = min{ le[v], le[u] - duration of <v,u>}
   - Decrease the out-degree of v
3. Repeat the steps until all vertices are visited
   - For the vertex w that has no predecessor, le [w] = ee [w] !

· 記誰指向它

Critical-path analysis can be carried out with AOV network

Free float : amount of time that a task can be delayed without causing a delay to the earliest start time of any immediately following activities

- earilest finish time & latest finish time for each activity

Maximum Flow Problem

We are given a flow network G with source (出發点) s and sink (終点) t, and we wish to find a flow of maximum value from s to t

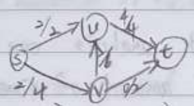
- Single source single sink maximum flow problem
- Maximum-flow min-cut theorem
  ↳ 畫一條線使所有点分左、右(同方向加總減反向)

A simplified model of Soviet railway traffic flow

- Formulated by T.E. Harris 1954

Ford-Fulkerson algorithm, 1955

- Residual graph (剩下)

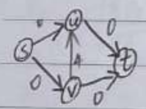


- residual capacity: $c_r(u,v) = c(u,v) - f(u,v)$, $c_r(v,u) = c(v,u) - f(v,u)$

Edmonds-Karp algorithm, 1972

- Heuristic to find augmenting (放大) path

Residual Graph

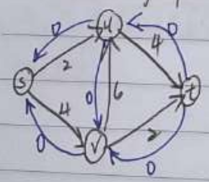


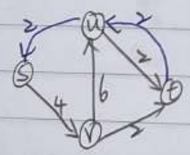
$s \to u \to t$: $c(s,u)=2$, $c(u,t)=4 \to flow(u,v)=2$

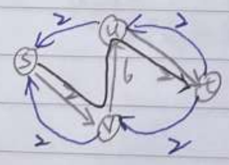
$s \to v \to t$: $c(s,v)=4$, $c(v,t)=2 \to flow(u,v)=2$

$s \to v \to u \to t$: $flow(u,v) = \min\{2, 6, 2\} = 2$

Residual graph

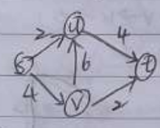






Ford-Fulkerson algorithm

1. Initialize $c_r(u,v)$ for every edge
2. Find a path P from s to t $\ni c_r(u,v) > 0 \ \forall (u,v) \in P$
3. $c_r(P) = \min\{c_r(u,v) : (u,v) \in P\}$;
4. For each edge $(u,v) \in P$
   - $c_f(u,v) = c_f(u,v) - c_f(P)$
   - $c_f(v,u) = c_f(v,u) + c_f(P)$ ↳ 藍線

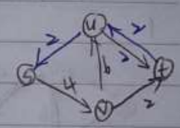


起 / 終

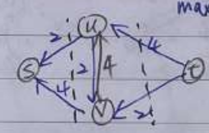
c_f	s	u	v	t
s	0	~~2~~ 0	4	0
u	2	0	0	~~4~~ 2
v	0	6	0	2
t	0	2	0	0

P: $s \to u \to t$

$c_f(P) = 2$



做到剩藍色的邊

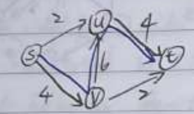


max flow
$4 + 2 = 6$

No:
Date: /

Edmonds-Karp algorithm
1. Initialize $C_f(u,v)$ for every edge
2. Find a path P from s to t by a heuristic → 1. max-capacity first; 2. breadth first 寬度優先 (保證短的先走)
3. $C_f(P) = \min\{C_f(u,v) : (u,v) \in P\}$
4. For each edge $(u,v) \in P$
   - $C_f(u,v) = C_f(u,v) - C_f(P)$
   - $C_f(v,u) = C_f(v,u) + C_f(P)$

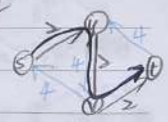
1. 只是猜測



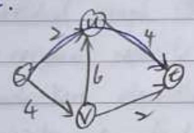
C_f	s	u	v	t
s	0	2	~~4~~ 0	0
u	0	0	0	~~4~~ 0
v	0	~~2~~ 6	0	2
t	0	0	0	0

P: s→v→u→t

$C_f(P) = 4$

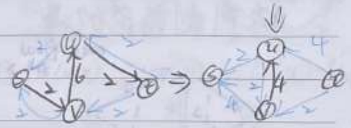


2.



P: s→v→t

$C_f(P) = 2$

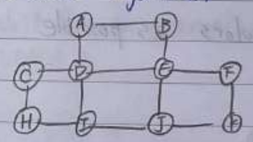


No:
Date: /

Eulerian circuit (Euler tour)
- Find a tour that would pass each edge exactly once and finally return to the starting vertex

Hamilton circuit
- Find a tour that would visit each vertex exactly once and finally return to the starting vertex
- Decision problem (NP-complete)

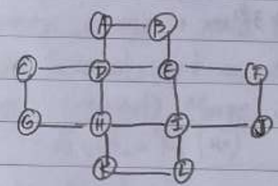
DFS-based Algorithm



Path: A
B
E {F, J} ⇒ A
D {C, I}
A

A
B
E
F
K
J {I}
E
D {C, I}
A

Not exist!



{F, I}
A B E D A
{H, L}
A B E F J I E D A
A B E F I H D C G H K L I E D A

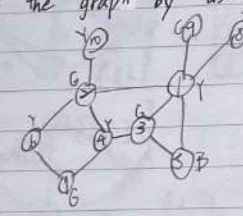
TSP on the Web: DEMOs
- Brute-force algorithm 暴力 (答案)
- Greedy algorithm 快但效率差 (速度)
- Branch-and-bound algorithm (擇中)
  可能做的選擇(分支) ↳ 走法的總距離 (上限)

No:
Date: /

Graph Coloring Problem

· Vertex Coloring (Edge Coloring)
· Sequential ordering algorithms (顏色順序)
- Heuristics for a specific ordering of vertices
  · No guarantee on using the least number of colors
- Welsh-Powell algorithm (greedy coloring)
  ∘ max-degree first

Color the graph by using as less colors as possible



偶 4个点 → 2色
奇 3个点 → 3色

Bi-connected Graph

Articulation poin 關節
- 但被拿掉就會 disconnected

Bi-connected graph
→ A connected graph that has no articulation point
(双重保障) (任何点拿掉都不會 disconnected)

Finding the articulation points
- Graph traversal algorithm
- DFS-tree based algorithm → 所能走到的最小數字
  退回時回傳編號 (沒路可走的就不會是 樞紐点)
  子 > 父 → 是 (只要一个是就是樞紐点

No:
Date: /

Secondary Storage

Main Memory vs. Secondary Storage
- CPU time vs. I/O time (seek + latency + transfer)
  上 下 愈少愈好 固定

Sequential Access vs. Direct Access (Random Access)
- Block access → organize file as user-defined blocks
File manger in OS supports
- Cluster : 檔案真正存的東西 (a number of contiguous sectors)
I/O Processor
- Wait for an external data path to become available (DMA)

External Sort

secondary storage + main memory
Step1. Internal sort on each block
Step2. (external) Merge sort
$\overset{10}{B_1} + \overset{10}{B_2} = \overset{20}{R_1}$ (40)
$R_1 + R_2 = S_1$

2-way Merge

64 runs → 32, 16, 8, 4, 2, 1 → $\log_2 64 = 6$ passes (+1 = 讀寫次數)
第一次的
16 runs → 8, 4, 2, 1 → $\log_2 16 = 4$ passes
· A k-way merge on m runs needs $\log_k m$ passes
· Higher-order merge can reduce I/O time
4-way merge
- 64 runs → 16, 4, 1 → $\log_4 64 = 3$ passes
- 16 runs → 4, 1 → $\log_4 16 = 2$ passes

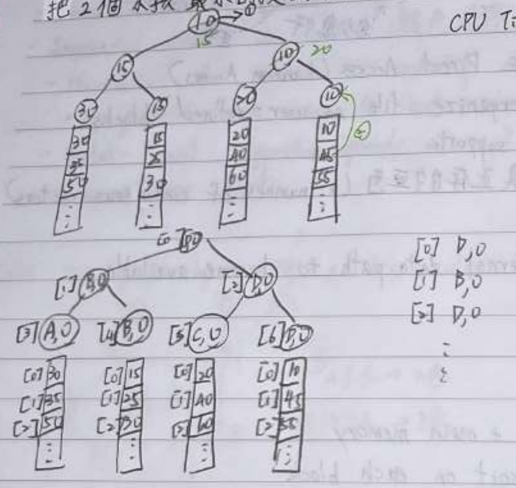
No:
Date: /

Selection Tree

把2個小孩最小的提出來

$k \rightarrow \log k$

CPU Time



File Structures

Record (object)

- Field

$\boxed{\begin{array}{l}\cdot\text{name}\\ \cdot\text{value}\end{array}}$

1. Force the field into a predictable length
2. Begin each field with a length indicator
3. Separate the fields with delimiters (不適合做資料交換)
4. Use a "fieldname = value" expression to identify each field and its content (檔案大)

No:
Date: /

Records

1. Requiring that the records be a predictable number of bytes in length.
2. Requiring that the records be a predictable number of fields in length.
3. Beginning each record with a length indicator consisting of a count of the number of bytes that the record contains.
4. Placing a delimiter (間隔符號) at the end of each record to separate it from the next record.
5. Using a second file (index) to keep track of the beginning byte address for each record.

(Header record length) + $(i-1)$ * (record length)

offset $= 1-(2-1)^* 100 = 101$ bytes

Deletion

Deletion of record RRN=2

1. move records 3, 4 to 2, 3
2. move record 4 to 2
3. do not move any record, but link all free records as a free list

Free List

- List head
  - Stored in the file header
  - Keep the RRN of one deleted record
- Use one field of the deleted record to keep the RRN of the next deleted record
- Regard these RRNs (offset) as pointers in the file

No:
Date: /

Key Sort

Open input file as IN_FILE
Create output file OUT_FILE
Read header record from IN_FILE
REC_COUNT = record count stored in head record
// read all the records in sequence
for i=1 to REC_COUNT
  do loop1
sort(KEY_ARRAY, REC_COUNT)
// repeatedly read and write each record
for i=1 to REC_COUNT
  do loop2
Close IN_FILE and OUT_FILE

loo1
1. read record from IN_FILE into BUFFER (in order)
2. KEY_ARRAY[i].KEY = extract key from BUFFER
3. KEY_ARRAY[i].RRN = i

loop2
1. j = KEY_ARRAY[i].RRN
2. seek in IN_FILE to the record with RRN=j
3. read record from IN_FILE into BUFFER
4. write BUFFER to OUT_FILE (in order) → BUFFER size = record length

No:
Date: /

Secondary Index

B-tree Index

Similar to 2-3 tree, instead of 2-3-4 tree
- Split when the node to insert is full ($m-1$ keys)
- Among the m keys (sorted), move the $[m/2]$-th key to the parent node (upward recursion)

m 愈大 樹高愈低

Insertion
1. Add the data record → get the location in file
2. Add the index entry

Deletion
1. Remove the index entry → get the location in file
2. Remove the data record

B^*-tree

delayed split + better space utilization
- Root has 2~m children
- The other node has $[(2m-1)/3]$ ~ m children
- Failure nodes are at the same level

$m=6$: $[(2m-1)/3] = 4$ → 3~5 keys

上面放的是索引

B^+-tree

fixed-size node + range query
- Root has 2~m children
- The other non-leaf node has $[m/2]$ ~ m children
- The leaf has $[(m-1)/2]$ ~ m-1 keys
- Failure nodes are at the same level

回答範圍問題有效

No:

Date: /

Hash Indexing Methods

Static Hash

- Fixed-length hash table

Extensible Hash

- Hash table size is doubled if ncessary (放不下就x2)

Linear Hash

- Hash table size grows linearly (放不下就+1)