Tamkang Universty Software Engineering Group
淡 江 軟 體 工 程 實 驗 室
http://www.tkse.tku.edu.tw/

演算法

教材： Data Structures & Algorithms

Michael T. Goodrich等 新月圖書代理

Reporter : Ying-Hong Wang

E-mail :inhon@mail.tku.edu.tw

Date : May 31, 2015

課前說明

- 上課用書
  - Data Structures and Algorithms in Java 4/e Goodrich著 John Wiley出版 新月圖書代理
  - 請尊重智慧財產權，勿非法影印使用教科書與參考書籍及使用盜版軟體
- 上課方式
  - 將採多元授課方式，包含板書、 投影片講授、團隊討論或個人問答、團體或個人報告等翻轉或PBL教學，須牢記本課程的成績卡座號
- 上課規定
  - 進入手機改設震動或關機、不要私下講話 (私下講話者扣學期成績)
  - 上課時間僅可喝水、不可吃東西，用餐請利用課間(課前、課後)完成
- 考試方式
  - Closed book

課前說明

- 成績評定
  - 出席:15% 實習課:15% 作業+小考:40% 期中考:15% 期末考:15%
  - 出席成績：每週必點名一次，預計點名15次，每人每學期有3次免責權，缺席三次以上者才開始扣出席成績。相對地，全勤者學期成績另加3分。點名係用以區別出席上課與否，故無論任何原因缺席者，均不需要請假，亦不補點。
  - 遲到、早退者，列入紀錄，每累計達二次者，視同缺席一次
  - 一般作業可以遲交，逾時每24小時內扣10分，扣至0分為止
  - 隨堂作業、小考、課堂問答、指定報告等不受理補交或補考，敬請隨時攜帶課本、講義、筆記等。隨堂作業與小考僅收A4格式紙，敬請隨身準備，非指定格式，視同未交。
  - 期末考比校訂時間提前一週，含實習課內容。

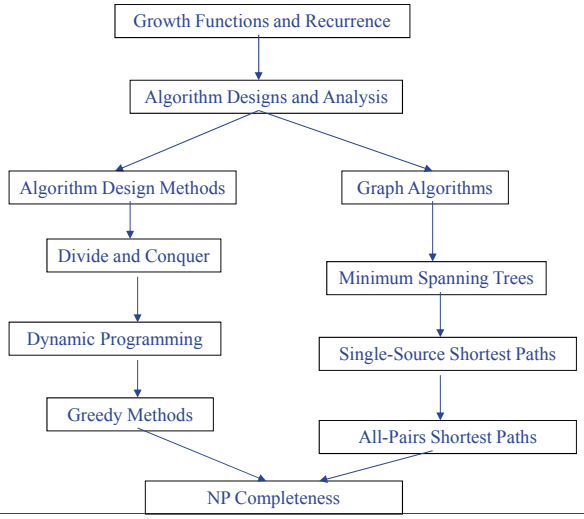
課前說明

- 課前測驗
  - 本週五(3/06)辦理課前測驗
  - 0830起，測驗時間60分鐘
  - 本測驗不計分、無補測
  - 但是，未參加者需退選本課程或不可加選本課程
  - 測驗結果將作為分組依據、無分組無分數
  - 分組團體由授課教師分配，不可自行選換組
  - 課程進行中，個人表現與團隊表現都會左右每一位同學的全學期成績
  - 每個人的成績都是依賴全學期一點一滴的累加
  - 請跟大一的自己比較，知識要有增長、技能要有精進

課程綱要

- Introduction
- Graphs
- Preview of the Algorithm Designs and Analysis
- Divide and Conquer
- Dynamic Programming
- Greedy Methods
- NP Completeness
- 實習課部分
  - Recurrences
  - Advanced Data Structure: Red-Black Tree, AVL-Tree, B-Tree, B^+ Tree
  - NP Complete Problems and Approaching Solutions

Learning Objects Architecture



Agenda

1. Introduction

2. Graphs

3. Preview of Algorithm Designs and Analysis

4. Divide and Conquer

5. Dynamic Programming

6. Greedy Methods

7. NP Completeness

Introduction

- Recall The Definition of Algorithm
- The Role of the Algorithms in Computer
- What kind of problem can be solved by algorithm?
- Example
- Analyzing Algorithms
- Designing Algorithms
- NP Problem

Recall the Definition of Algorithm

- An algorithm is a **finite** sequence of **unambiguous** instructions for solving a well-specified computational problem.
- Important Features:
  - Finiteness.
  - Definiteness.
  - Input.
  - Output.
  - Effectiveness.

Why to Study Algorithm

Develop thinking ability.

- problem solving skills.
  (algorithm design and application)
- formal thinking.
  (proof techniques & analysis)

The Role of the Algorithms in Computer

- An ***instance of a problem*** consists of all inputs needed to compute a solution to the problem.
- An algorithm is said to be ***correct*** if for every input instance, it halts with the correct output.
- A correct algorithm ***solves*** the given computational problem. An incorrect algorithm might not halt at all on some input instance, or it might halt with other than the desired answer.

What kind of problem can be solved by algorithm?

- The Human Genome Project
- The Internet Applications
- Electronic Commerce with Public-key cryptography and digital signatures
- Manufacturing and other commercial settings

Insertion sort

- **Example:** Sorting problem
  - Input: A sequence of n numbers
  - Output: A permutation of the input sequence such that .
- The number that we wish to sort are known as the *keys*.

Insertion sort

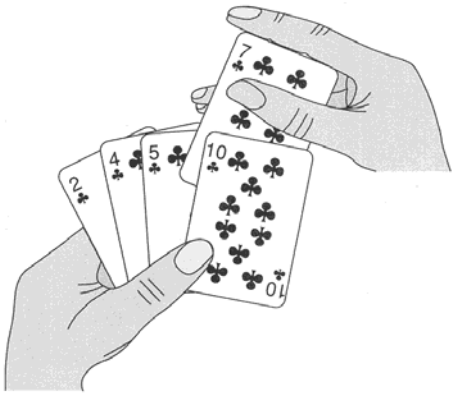


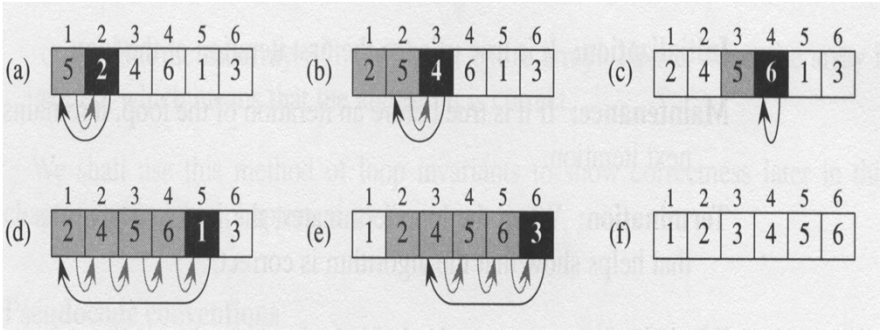
Figure 2.1 Sorting a hand of cards using insertion sort.

Pseudocode

Insertion sort

```
Insertion-sort(A)
1  for j ←2 to length[A]
2    do key←A[j]
3      ∗Insert A[j] into the sorted sequence A[1..j-1]
4       i ←j - 1
5       while i>0 and A[i]>key
6             do A[i+1] ←A[i]
7                i ←i - 1
8       A[i +1] ←key
```


The operation of Insertion-Sort



延伸閱讀

- 何謂Algorithm？
  - http://www.csie.ntnu.edu.tw/~u91029/Algorithm.html
  - http://en.wikipedia.org/wiki/Algorithm
  - https://www.khanacademy.org/computing/computer-science/algorithms
  - http://whatis.techtarget.com/definition/algorithm
- Video materials
  - https://www.youtube.com/watch?v=S-ws2W6UbPU
  - https://www.youtube.com/watch?v=JPyuH4qXLZ0&list=PL14288029B0AEEA6E
  - https://www.youtube.com/watch?v=HtSuA80QTyo&list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb

Analyzing Algorithms

- Time Complexity
- Space Complexity
- Worst-case and Average-Case Analysis
- Order of Growth

Worst, Average, and Best-case Complexity

- **Worst-case Complexity**
  - **Maximum** steps the algorithm takes for any possible input.
  - Most tractable measure.
- **Average-case Complexity**
  - **Average** of the running times of all ***possible*** **inputs.**
  - Demands a definition of probability of each input, which is usually difficult to provide and to analyze.
- **Best-case Complexity**
  - **Minimum** number of steps for any possible input.
  - Not a useful measure. Why?
- **Stable and Unstable**

Worst-case and average-case analysis

- Usually, we concentrate on finding only on the ***worst-case running time***
- Reason:
  - It is an upper bound on the running time
  - The worst case occurs fair often
  - The average case is often as bad as the worst case. For example, the insertion sort. Again, quadratic function.

Order of growth

- In some particular cases, we shall be interested in ***average-case***, or ***expect*** running time of an algorithm.

- It is the ***rate of growth***, or ***order of growth***, of the running time that really interests us.

Order of growth

- Principal interest is to determine
  - how running time grows with input size – **Order of growth**.
  - the running time for large inputs – **Asymptotic complexity**.
- In determining the above,
  - **Lower-order terms and coefficient of the highest-order term are insignificant.**
  - **Ex: In $7n^5+6n^3+n+10$, which term dominates the running time for very large n?**
- Complexity of an algorithm is denoted by the highest-order term in the expression for running time.
  - Ex: $O(n)$, $\Theta(1)$, $\Omega(n^2)$, etc.
  - Constant complexity when running time is independent of the input size – denoted $O(1)$.
  - Linear Search: **Best case $\Theta(1)$, Worst and Average cases: $\Theta(n)$.**
- More on O, Θ, and Ω in next class. Use Θ for the present.

Comparison of Algorithms

- Complexity function can be used to compare the performance of algorithms.
- Algorithm *A* is more efficient than Algorithm *B* for solving a problem, if the complexity function of *A* is of lower order than that of *B*.
- Examples:
  - **Linear Search** – $\Theta(n)$ vs. **Binary Search** – $\Theta(\lg n)$
  - **Insertion Sort** – $\Theta(n^2)$ vs. **Quick Sort** – $\Theta(n \lg n)$

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Comparison of Algorithms

- **Multiplication**
  - classical technique: $O(nm)$
  - divide-and-conquer: $O(nm^{\ln 1.5}) \sim O(nm^{0.59})$

    For operands of size 1000, takes 40 & 15 seconds respectively on a Cyber 835.
- **Sorting**
  - insertion sort: $\Theta(n^2)$
  - merge sort: $\Theta(n \lg n)$

    For 10^6 numbers, it took 5.56 hrs on a supercomputer using machine language and 16.67 min on a PC using C/C++.

延伸閱讀

- 何謂Asymptotic Notations ?
  - http://en.wikibooks.org/wiki/Data_Structures/Asymptotic_Notation
  - http://content.edu.tw/senior/computer/ks_ks/book/algodata/algorithm/algo5.htm
  - http://mathworld.wolfram.com/AsymptoticNotation.html
  - https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation
- Video materials
  - https://www.youtube.com/watch?v=whjt_N9uYFI
  - http://mathworld.wolfram.com/AsymptoticNotation.html
  - https://www.youtube.com/watch?v=aGjL7YXI31Q
  - https://www.youtube.com/watch?v=6Ol2JbwoJp0

Designing Algorithms

Classification of Algorithm Design Methods

- Incremental Approach
- Divide and Conquer
- Randomization (Randomized Algorithm)
- Linear Programming
- Dynamic Programming
- Greedy method
- Brute-force
- Backtracking
- Branch and Bound

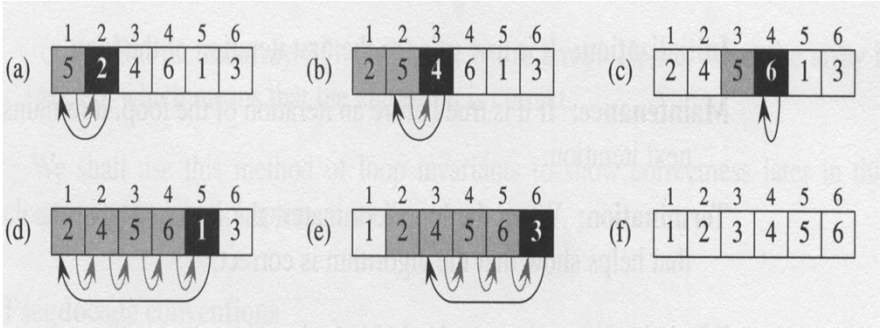
Designing Algorithms-Examples

- Incremental Approach
- Divide-and-Conquer Approach
- Analyzing Divide-and-Conquer Algorithms

Designing algorithms-Examples

- There are many ways to design algorithms:
- Example of a Sorting Problem
  - **Incremental approach**: insertion sort
  - **Divide-and-conquer**: merge sort
    - recursive:
      - *divide*
      - *conquer*
      - *combine*

Recall Insertion Sort



Divide and Conquer

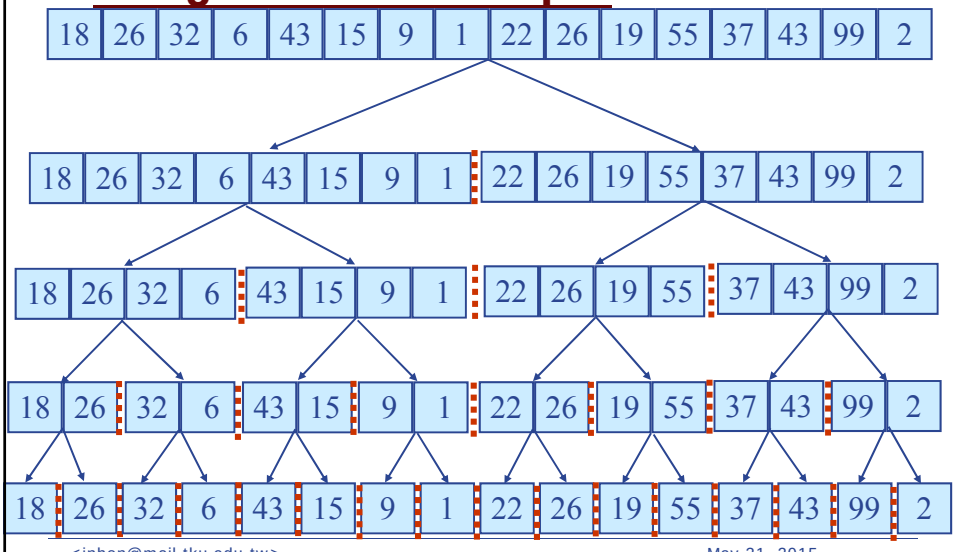
- Recursive in structure
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - ***Combine*** the solutions to create a solution to the original problem

An Example: Merge Sort

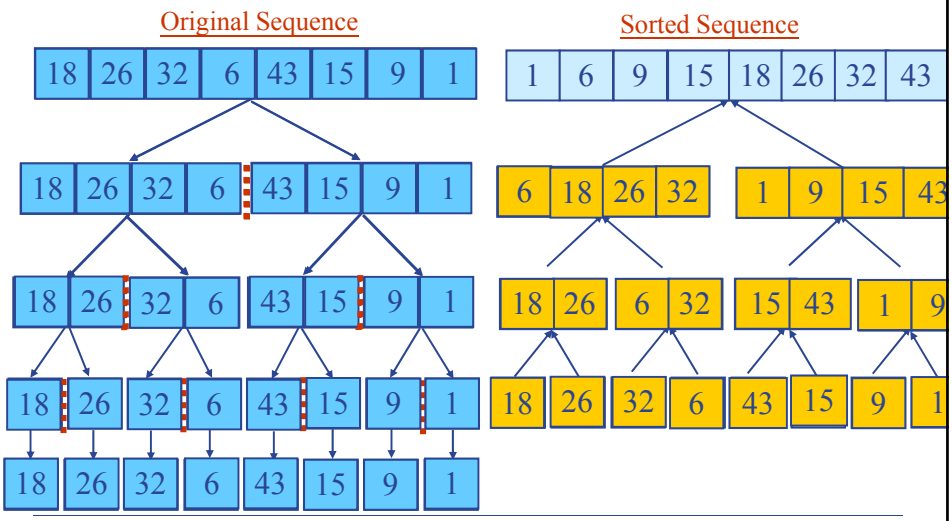
***Sorting Problem*:** Sort a sequence of *n* elements into non-decreasing order.

- ***Divide*:** Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each
- ***Conquer:*** Sort the two subsequences recursively using merge sort.
- ***Combine*:** Merge the two sorted subsequences to produce the sorted answer.

Merge Sort – Example



Merge Sort – Example



NP Problem

- A problem is assigned to the NP (nondeterministic polynomial time) class if it cannot be solved in polynomial time.
- A problem is said to be NP-hard if an algorithm for solving it can be translated into one for solving any other NP-problem. It is much easier to show that a problem is NP than to show that it is NP-hard.
- A problem which is both NP and NP-hard is called an NP-complete problem.

Agenda

1. Introduction

2. Graphs

3. Preview of Algorithm Designs and Analysis

4. Divide and Conquer

5. Dynamic Programming

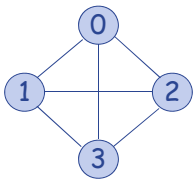
6. Greedy Methods

7. NP Completeness

Graphs

- Recall the Representation of Graphs
- Basic Algorithms of Graphs
- Biconnected Components
- Minimum Cost Spanning Trees
- Single-Source Shortest Paths
- All-Pairs Shortest Paths
- Topological Sort
- Strongly connected components

Representation of Graphs



$V(G_1)$ = {0, 1, 2, 3}

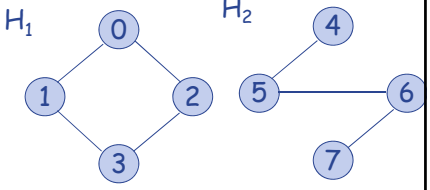
$E(G_1)$ = {(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)}

(a) G_1

$V(G_2)$ = {0, 1, 2}

$E(G_2)$ = {<0, 1>, <1, 0>, <1, 2>}

(b) G_2



$V(G_3)$ = {0, 1, 2, 3,4, 5, 6, 7}

$E(G_3)$ = {(0, 1), (0, 2), (1, 3), (2, 3), (4, 5), (5, 6), (6, 7) }

(c) G_3

Adjacency Matrices

	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

(a) G_1

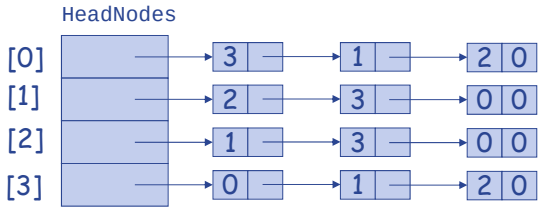
	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

(b) G_3

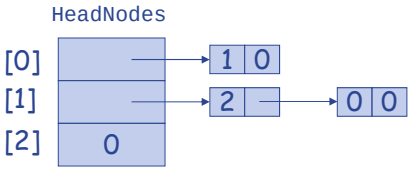
	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0
2	1	0	0	1	0	0	0	0
3	0	1	1	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	1	0	1	0
6	0	0	0	0	0	1	0	1
7	0	0	0	0	0	0	1	0

(c) G_3

Adjacent Lists



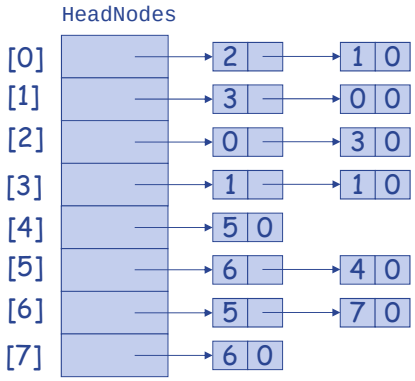
(a) G_1



(b) G_2

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Adjacent Lists



(c) G_3

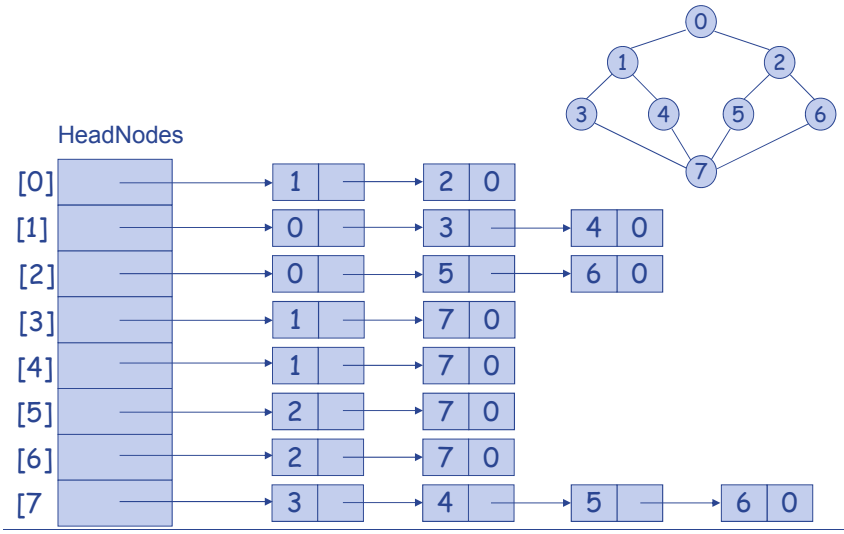
Weighted Edges

- Very often the edges of a graph have weights associated with them.
  - distance from one vertex to another
  - cost of going from one vertex to an adjacent vertex.
  - To represent weight, we need additional field, weight, in each entry.
  - A graph with weighted edges is called a *network*.

The Basic Algorithms of Graphs

- A general operation on a graph G is to visit all vertices in G that are reachable from a vertex v.
  - Depth-first search
  - Breath-first search

Graph G and Its Adjacency Lists



Depth-First Search

- Starting from vertex, an unvisited vertex w adjacent to v is selected and a depth-first search from w is initiated.
- When the search operation has reached a vertex u such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w again.
- The above process repeats until no unvisited vertex can be reached from any of the visited vertices.

Algorithm of DFS

```
tree T = empty //T is the Spanning Tree

DFS(vertex v) {
    visit(v);
    for (each neighbor w of v)
        if (w is unvisited) {
            DFS(w);
            add edge (v, w) to tree T
        }
}
```


Analysis of DFS

- If G is represented by its adjacency lists, the DFS time complexity is O(e).
- If G is represented by its adjacency matrix, then the time complexity to complete DFS is $O(n^2)$.

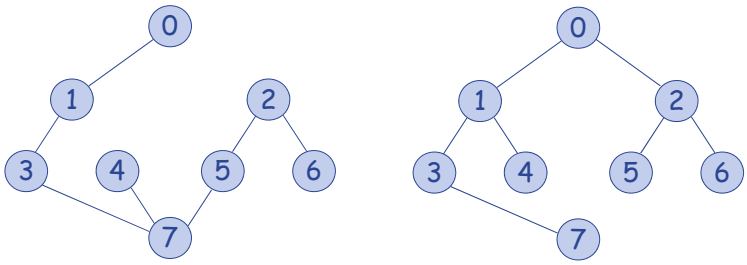
Breath-First Search

- Starting from a vertex v, visit all unvisited vertices adjacent to vertex v.
- Unvisited vertices adjacent to these newly visited vertices are then visited, and so on.
- If an adjacency matrix is used, the BFS complexity is $O(n^2)$.
- If adjacency lists are used, the time complexity of BFS is O(e).

Algorithm of BFS

```
list L = empty //L is a Queue
tree T = empty //T is the Spanning Tree
Unmark all vertices
BFS(vertex x) {
   choose some starting vertex x
   mark  x; Add x to L; Add x to T;
   while (L is nonempty) {
        choose vertex v from the front of L
        visit v for each unmarked neighbor w
        mark w;
        add w to the end of L
        add edge (v, w) to T
    }
}
```

Depth-First and Breath-First Spanning Trees



(a) DFS (0) spanning tree (b) BFS (0) spanning tree

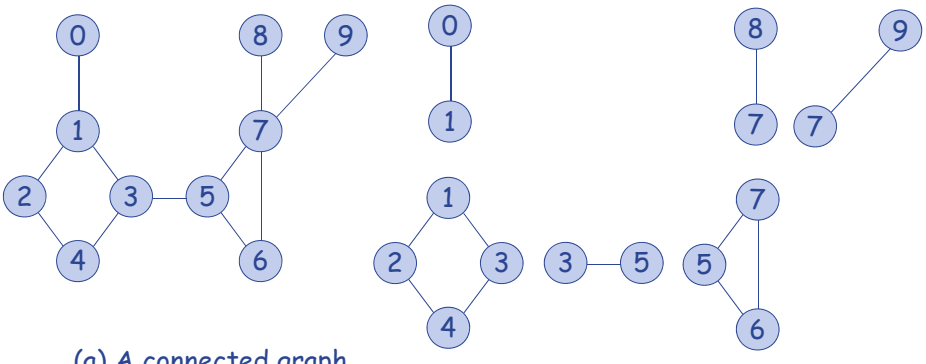
延伸閱讀

- http://en.wikipedia.org/wiki/Depth-first_search
- http://en.wikipedia.org/wiki/Breadth-first_search
- http://www.csie.ntnu.edu.tw/~u91029/Graph.html#4
- http://www.csie.ntnu.edu.tw/~u91029/Graph.html#5
- https://www.youtube.com/watch?v=zLZhSSXAwxI
- https://www.youtube.com/watch?v=AfSk24UTFS8
- https://www.youtube.com/watch?v=s-CYnVz-uh4

Biconnected Components

- Definition: A vertex v of G is an articulation point iff the deletion of v, together with the deletion of all edges incident to v, leaves behind a graph that has at least two connected components.
- Definition: A biconnected graph is a connected graph that has no articulation points.
- Definition: A biconnected component of a connected graph G is a maximal biconnected subgraph H of G. By maximal, we mean that G contains no other subgraph that is both biconnected and properly contains H.

A Connected Graph and Its Biconnected Components



(a) A connected graph

(b) Its biconnected components

延伸閱讀 – Self Study Unit

- http://en.wikipedia.org/wiki/Biconnected_component
- http://web.thu.edu.tw/johnaxer/www/algorithm/ppt/chapter6.ppt
- http://www.csie.ntu.edu.tw/~hsinmu/courses/_media/dsa_13spring/horowitz_306_311_biconnected.pdf
- http://www.csie.ntnu.edu.tw/~u91029/Component.html#3
- https://www.youtube.com/watch?v=Jl7icfFUCs4
- https://www.youtube.com/watch?v=Ss5WikSTtLg

Minimum Cost Spanning Tree

- The *cost* of a spanning tree of a weighted, undirected graph is the sum of the costs (weights) of the edges in the spanning tree.
- A *minimum-cost spanning tree* is a spanning tree of least cost.
- Three greedy-method algorithms available to obtain a minimum-cost spanning tree of a connected, undirected graph.
  - Kruskal's algorithm ✡
  - Prim's algorithm ✡
  - Sollin's algorithm

Kruskal's Algorithm

- Kruskal's algorithm builds a minimum-cost spanning tree T by adding edges to T one at a time.
- The algorithm selects the edges for inclusion in T in nondecreasing order of their cost.
- An edge is added to T if it does not form a cycle with the edges that are already in T.
- Theorem 6.1: Let G be any undirected, connected graph. Kruskal's algorithm generates a minimum-cost spanning tree.

Steps of Kruskal's Algorithm(1)

- Set i=1 and let E_0={}
- Select an edge e_i of minimum value not in E_i-1 such that T_i=<E_i-1 with {e_i} >is acyclic and define E_i=E_i-1 with {e_i}. If no such edge exists, Let T=<E_i>and stop.
- Replace i by i+1. Return to Step 2.
- The time required by Kruskal's algorithm is O(|E|log|V|)

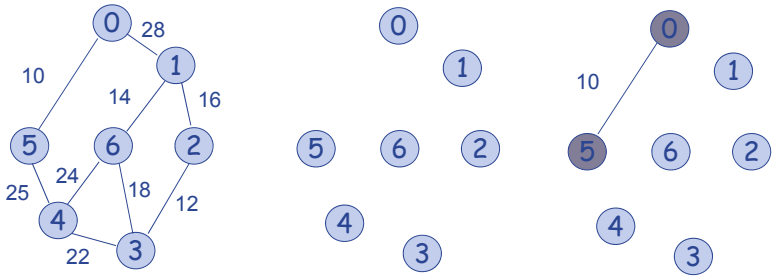
Steps of Kruskal's Algorithm(2)

```
1  T = Φ;
2  while ( ( T包含的邊少於n – 1個) && (E不是空的) ) {
3     從E中選一個花費最低的邊 (v, w);
4     從E中刪除 (v, w);
5        if ( (v, w) 不會在T中產生迴路 ) 把 (v, w) 加到T;
6        else 忽略 (v, w);
7  }
8  if ( T包含的邊不滿n – 1個 ) cout << "no spanning tree" << endl;
```


Steps of Kruskal's Algorithm(3)

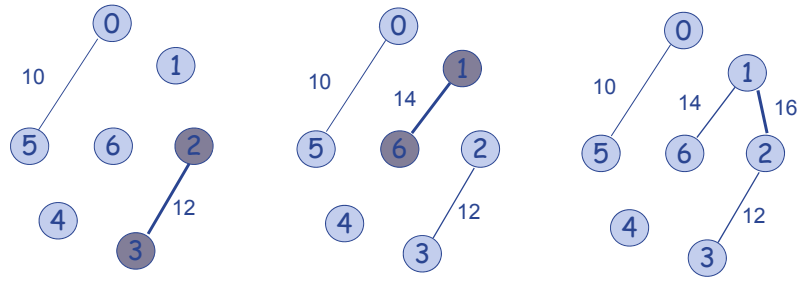
```
1 function Kruskal(G)
2    for each vertex v in G do
3       Define an elementary cluster C(v) ← {v}.
4       Initialize a priority queue Q to contain all edges in G, using the weights as keys.
5       Define a tree T ← Ø //T will ultimately contain the edges of the MST
6       // n is total number of vertices
7       while T has fewer than n-1 edges do
8          // edge u,v is the minimum weighted route from/to v
9          (u,v) ← Q.removeMin()
10         // prevent cycles in T. add u,v only if T does not already contain an edge consisting of u and v.
           // Note that the cluster contains more than one vertex only if an edge containing a pair of
           // the vertices has been added to the tree.
11         Let C(v) be the cluster containing v, and let C(u) be the cluster containing u.
12         if C(v) ≠ C(u) then
13            Add edge (v,u) to T.
14            Merge C(v) and C(u) into one cluster, that is, union C(v) and C(u).
15 return tree T
```

Stages in Kruskal's Algorithm(1)



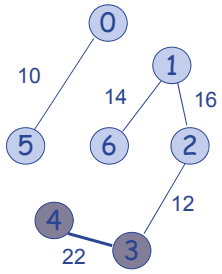
(a) (b) (c)

Stages in Kruskal's Algorithm(2)

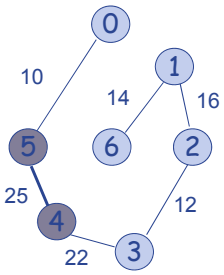


(d) (e) (f)

Stages in Kruskal's Algorithm(3)



(g)



(h)

Prim's Algorithm

- Similar to Kruskal's algorithm, Prim's algorithm constructs the minimum-cost spanning tree edge by edge.
- The difference between Prim's algorithm and Kruskal's algorithm is that the set of selected edges forms a tree at all times when using Prim's algorithm while a forest is formed when using Kruskal's algorithm.
- In Prim's algorithm, a least-cost edge (u, v) is added to T such that T $\cup$ {(u, v)} is also a tree. This repeats until T contains n-1 edges.

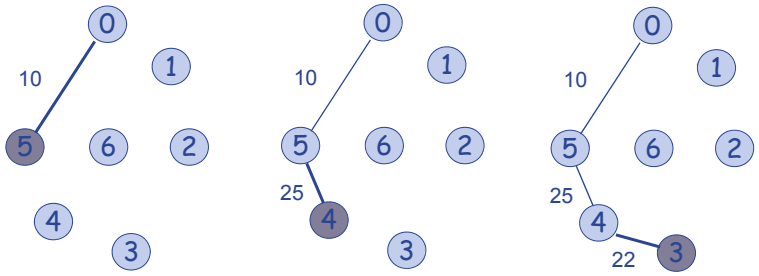
Steps of Prim's Algorithm(1)

- Set i=0, S_0= {u_0=s}, L(u_0)=0, and L(v)=infinity for v <> u_0. If |V| = 1 then stop, otherwise go to step 2.
- For each v in V\S_i, replace L(v) by min{L(v), d_v^{ui}}. If L(v) is replaced, put a label (L(v), u_i) on v.
- Find a vertex v which minimizes {L(v): v in V\S_i}, say u_i+1.
- Let S_i+1 = S_i with {u_i+1}.
- Replace i by i+1. If i=|V|-1 then stop, otherwise go to step 2.
- The time required by Prim's algorithm is O($|V|^2$). It will be reduced to O(|E|log|V|) if heap is used to keep {v in V\Si : L(v) < infinity}.

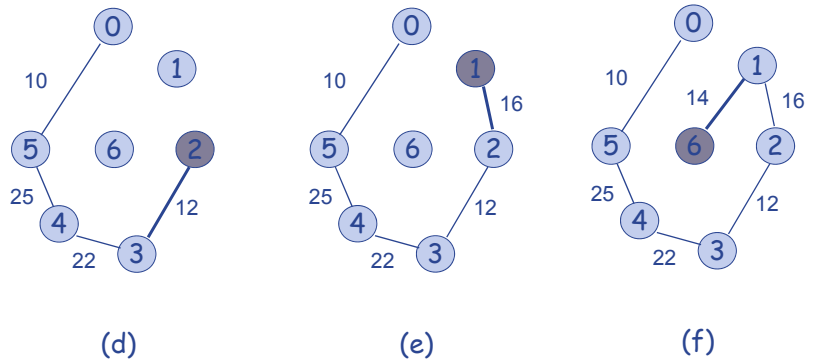
Steps of Prim's Algorithm(2)

```
1  // 假設G至少有一個頂點
2  TV = {0}; // 從頂點 0、沒有邊的狀態開始
3  for (T = Φ；T包含的邊不滿n – 1個；把 (u, v) 加入T)
4  {
5     令 (u, v) 為花費最少的邊，使u ∈ TV且v ∉ TV;
6     if (沒有這種邊) break;
7     將v加入TV;
8  }
9  if ( T包含的邊不滿n – 1個) cout << "no spanning tree" << endl;
```

Stages in Prim's Alogrithm(1)



Stages in Prim's Alogrithm(2)



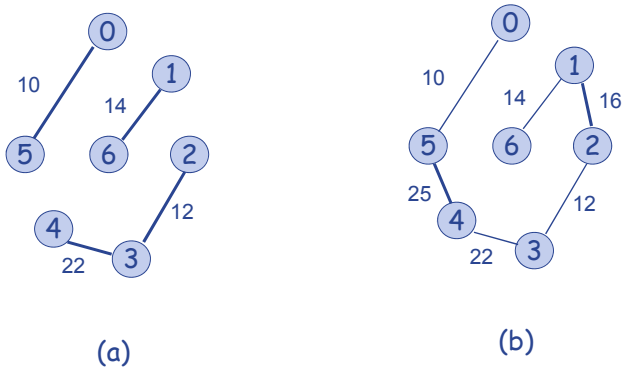
Sollin's Algorithm

- Contrast to Kruskal's and Prim's algorithms, Sollin's algorithm selects multiple edges at each stage.
- At the beginning, the selected edges and all the n vertices form a spanning forest.
- During each stage, an minimum-cost edge is selected for each tree in the forest.
- It's possible that two trees in the forest to select the same edge. Only one should be used.
- Also, it's possible that the graph has multiple edges with the same cost. So, two trees may select two different edges that connect them together. Again, only one should be retained.

Steps of Sollin's Algorithm

for each $i \in N$ **do**
  $S_i = \{i\}$.
end for
$T = \emptyset$ {These are the tree edges}
while $|T| < (n-1)$ **do**
  for each tree S_k **do**
    nearest-neighbor(S_k, i_k, j_k).
  end for
  for each tree S_k **do**
    if nodes i_k and j_k belong to different trees **then**
      merge(i_k, j_k)
      $T \leftarrow T \cup \{(i_k, j_k)\}$
    end if
  end for
end while

Stages in Sollin's Algorithm



延伸閱讀

- http://en.wikipedia.org/wiki/Shortest_path_problem
- https://www.cs.princeton.edu/~rs/AlgsDS07/15ShortestPaths.pdf
- https://www.cs.princeton.edu/~rs/AlgsDS07/15ShortestPaths.pdf
- https://www.youtube.com/watch?v=WN3Rb9wVYDY
- https://www.youtube.com/watch?v=dS1Di2ZHl4k
- https://www.youtube.com/watch?v=8Ls1RqHCOPw

Shortest Paths

- Usually, the highway structure can be represented by graphs with vertices representing cities and edges representing sections of highways.
- Edges may be assigned weights to represent the distance or the average driving time between two cities connected by a highway.
- Often, for most drivers, it is desirable to find the shortest path from the originating city to the destination city.

Single Source / All Destinations: Nonnegative Edge Costs

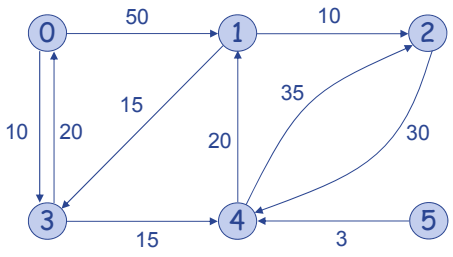
- Let S denotes the set of vertices to which the shortest paths have already been found.
  1) If the next shortest path is to vertex u, then the path begins at v, ends at u, and goes through only vertices that are in S.
  2) The destination of the next path generated must be the vertex u that has the minimum distance among all vertices not in S.
  3) The vertex u selected in 2) becomes a member of S.
- The algorithm is first given by Edsger Dijkstra. Therefore, it's sometimes called Dijkstra Algorithm.

Steps of Dijkstra Algorithm

```
1 function Dijkstra(G, w, s)
2     for each vertex v in V[G]     // Initializations
3         d[v] := infinity     // Known distance function from s to v
4         previous[v] := undefined
5     d[s] := 0     // Distance from s to s
6     S := empty set     // Set of all visited vertices
7     Q := V[G]     // Set of all unvisited vertices
8     while Q is not an empty set     // The algorithm itself
9         u := Extract_Min(Q)     // Remove best vertex from priority queue
10        S := S union {u}     // Mark it 'visited'
11        for each edge (u,v) outgoing from u
12            if d[u] + w(u,v) < d[v]     // Relax (u,v)
13                d[v] := d[u] + w(u,v)
14                previous[v] := u
```

The running time is $O(|V|.|E|)$.

Graph and Shortest Paths From Vertex 0 to all destinations(1)

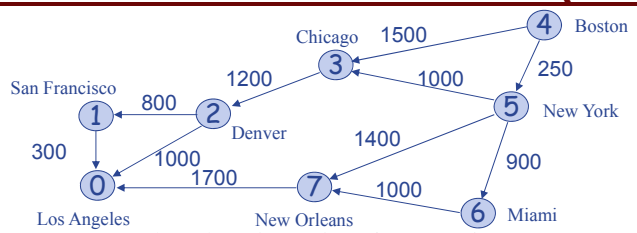


(a) Graph

Path	Length
1) 0, 3	10
2) 0, 3, 4	25
3) 0, 3, 4, 1	45
4) 0, 3, 4, 1, 2	55
5) 0, 5	+∞

(b) Shortest paths from 0

Graph and Shortest Paths From Vertex 4 to all destinations(2-1)



	0	1	2	3	4	5	6	7
0	0							
1	300	0						
2	1000	800	0					
3			1200	0				
4				1500	0	250		
5				1000		0	900	1400
6							0	1000
7	1700							0

Graph and Shortest Paths From Vertex 4 to all destinations(2-2)

iteration	S	Vertex selected	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial	--	---	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{4}	5	+∞	+∞	+∞	1250	0	250	1150	1650
2	{4,5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
3	{4,5,6}	3	+∞	+∞	2450	1250	0	250	1150	1650
4	{4,5,6,3}	7	3350	+∞	2450	1250	0	250	1150	1650
5	{4,5,6,3,7}	2	3350	3250	2450	1250	0	250	1150	1650
6	{4,5,6,3,7,2}	1	3350	3250	2450	1250	0	250	1150	1650
	{4,5,6,3,7,2,1}									

All-Pairs Shortest Paths

- In all-pairs shortest-path problem, we are to find the shortest paths between all pairs of vertices u and v, u ≠ v.
  - Use n independent single-source/all-destination problems using each of the n vertices of G as a source vertex. Its complexity is $O(n^3)$ (or $O(n^2 \log n + ne)$ if Fibonacci heaps are used).
  - On graphs with negative edges the run time will be $O(n^4)$. if adjacency matrices are used and $O(n^2e)$ if adjacency lists are used.

延伸閱讀

- http://en.wikipedia.org/wiki/Minimum_spanning_tree
- http://www.csie.ntnu.edu.tw/~u91029/SpanningTree.html#2
- http://www.csie.ntnu.edu.tw/~u91029/SpanningTree.html#3
- https://www.youtube.com/watch?v=8fJgkVpxbQg
- https://www.youtube.com/watch?v=G28gJ-uQREc
- https://www.youtube.com/watch?v=YyLaRffCdk4
- https://www.youtube.com/watch?v=k9jemw3SZe0

Strongly Connected Components

- In the directed graphs, a graph is said to be **strongly connected** if every vertex is reachable from every other vertex.
- The **strongly connected components** of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected.

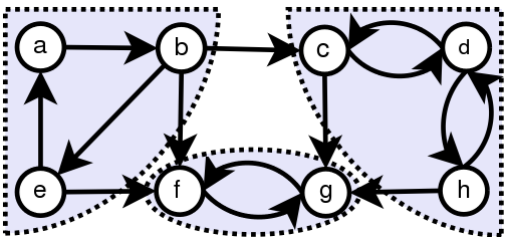
Self Study Unit

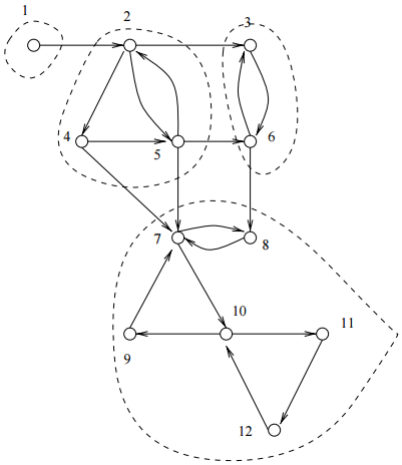
<inhon@mail.tku.edu.tw>

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Strongly Connected Components

Two examples





延伸閱讀

- http://en.wikipedia.org/wiki/Strongly_connected_component
- http://www.cs.berkeley.edu/~vazirani/s99cs170/notes/lec12.pdf
- http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/strongComponent.htm
- http://www.columbia.edu/~cs2035/courses/csor4231.F11/scc.pdf
- https://www.youtube.com/watch?v=J_Jl_r_Ua1Q
- https://www.youtube.com/watch?v=PZQ0Pdk15RA

Topological Sort

A topological sort of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

Dag: Directed Acicular Graph

定義：若在 AOV-network 中，V_i 是 V_j 的前行者，則在線性的排列中，V_i 一定在 V_j 的前面，此種特性稱之為拓樸排序 (topological sort) 。

AOV: Activity On Vertices

AOE: Activity On Edges

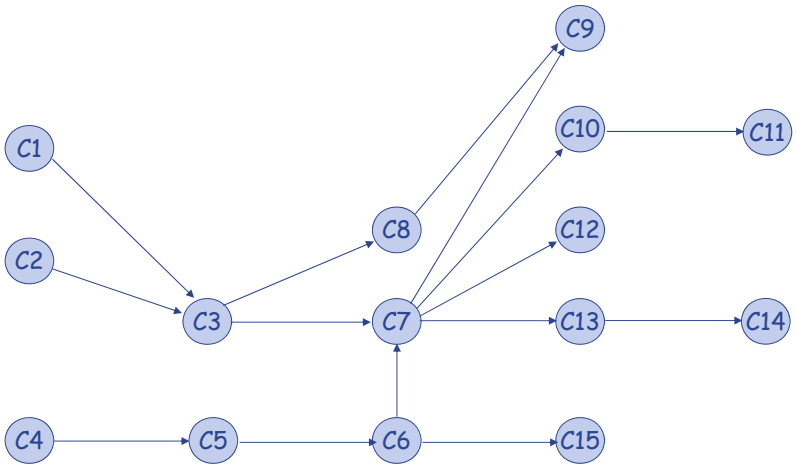
AOV and AOE

- Definition: A directed graph G in which the vertices represent tasks or activities and the edges represent precedence relations between tasks is an Activity-On-Vertex network or AOV network.
- Definition: A directed graph G in which the edges represent tasks or activities and the vertices represent precedence relations between tasks is an Activity-On-Edge network or AOE network.

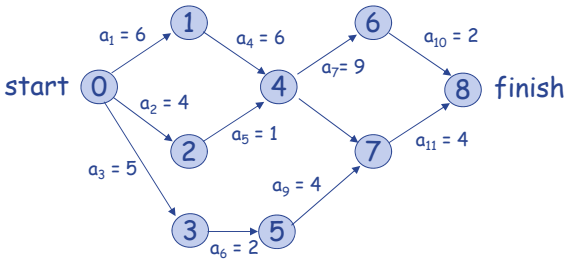
Example of AOV

Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

Example of AOV(Cont.)



Example of AOE



event	interpretation
0	Start of project
1	Completion of activity a_1
4	Completion of activities a_4 and a_5
7	Completion of activities a_8 and a_9
8	Completion of project

Topological Sort -Algorithms

尋找 AOV-network 拓樸排序的過程如下：

(1) 在 AOV-network 中任意挑選沒有前行者的節點。

(2) 輸出此頂點，並將該頂點所連接的邊刪除。

重覆步驟 (1) 及步驟 (2)，一直到全部的頂點皆輸出為止。

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Topological Sort -Algorithms

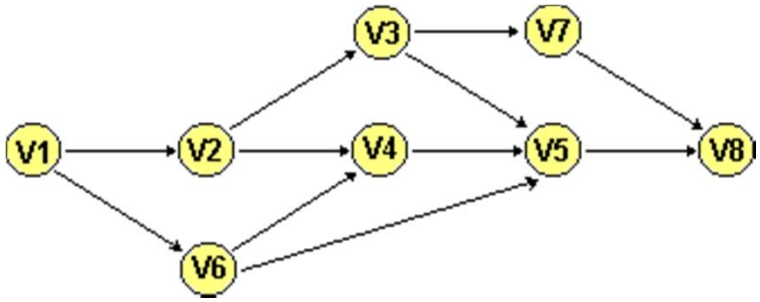
```
void topsort ( hdnodes , graph[] , int n )
    {
      int i , j , k , top ;
      node_pointer  ptr ;
      top = -1 ;
      for ( i = 0 ; i < n ; i++ )
        if ( !graph[i].count )
        {
          graph[i].count = top ;
          top = i ;
        }
      for ( i = 0 ; i < n ; i++ )
        if ( top == -1 )
        {
          fprintf ( stderr, "\n Network has a cycle.
                    Sort terminated. \n" ) ;
          exit (1) ;
        }
      else
      {
        j = top ;
        top = graph[top].count ;
        printf ( "v%d, " j ) ;
        for ( ptr = graph[j].link ; ptr ; ptr = ptr-
>link )
        {
          k = ptr -> vertex ;
          graph[k].count -- ;
          if ( !graph[k].count )
          {
            graph[k].count = top ;
            top =k ;
          }
        }
      }
}
```

The Time Complexity is O(n+e)

<inhon@mail.tku.edu.tw> May 31, 2015

Topological Sort -Example

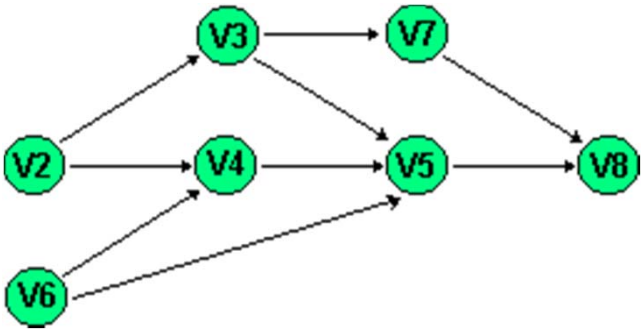
求下面 AOV-network 的拓樸排序



Topological Sort -Example

Step One:

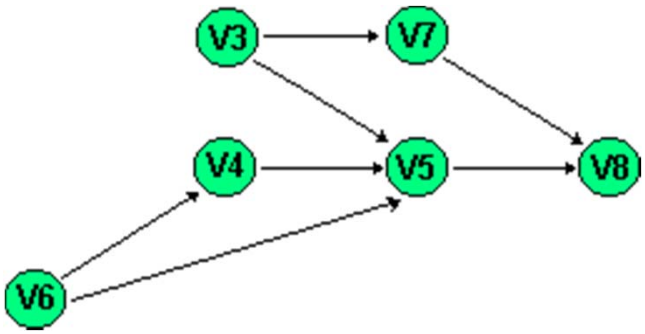
輸出 V1 ，並刪除 < V1,V2 > 與 < V1,V6 > 兩個邊



Topological Sort -Example

Step Two:

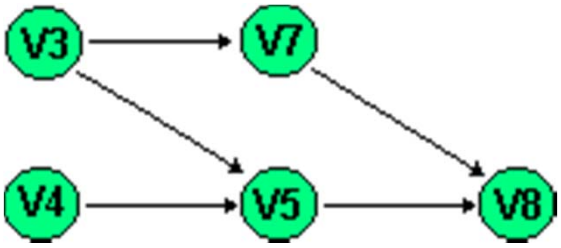
輸出 V2 ，並刪除 < V2,V3 > 與 < V2,V4 > 兩個邊



Topological Sort -Example

Step Three:

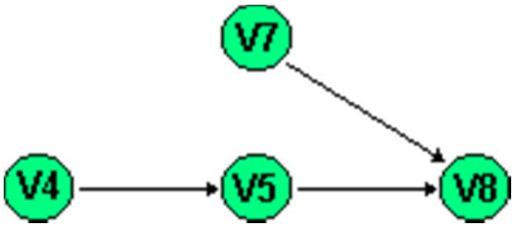
輸出 V6 ，並刪除 < V6,V4 > 與 < V6,V5 > 兩個邊



Topological Sort -Example

Step Four:

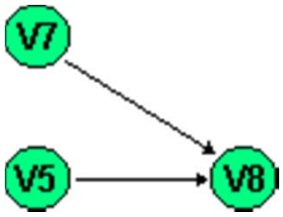
輸出 V3，並刪除 < V3,V7 > 與 < V3,V5 > 兩個邊



Topological Sort -Example

Step Five:

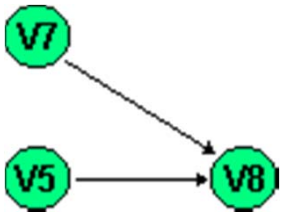
輸出 V4 ，並刪除 < V4,V5 >



Topological Sort -Example

Step Six:

輸出 V7 ，並刪除 < V7,V8 >



Topological Sort -Example

Step Seven:

輸出 V5 ，並刪除 < V5,V8 >

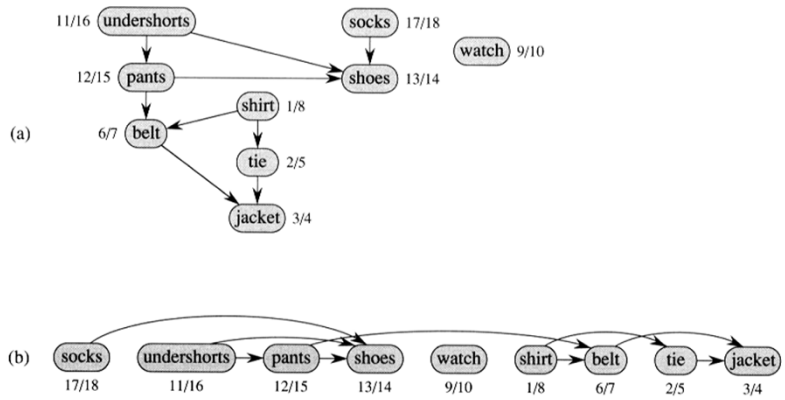
Step Eight:

輸出 V8

所得輸出順序為 1 , 2 , 6 , 3 , 4 , 7 , 5 , 8

相同的AOV，可能有二個以上的Topological Sort輸出

Topological Sort –Another Example



延伸閱讀

- http://en.wikipedia.org/wiki/Topological_sorting
- http://www.csie.ntnu.edu.tw/~u91029/DirectedAcyclicGraph.html
- http://www.csie.ntnu.edu.tw/~u91029/DirectedAcyclicGraph.html#2
- http://120.118.165.132/LMS/Content/C010/Tbank/Read/CH7/7-9/7-9.htm
- https://www.youtube.com/watch?v=jksMzq4LgfM
- https://www.youtube.com/watch?v=2E7tzF4ihvI
- https://www.youtube.com/watch?v=jHWGir3Jk8o

Agenda

Growth of Functions

Self Study Unit

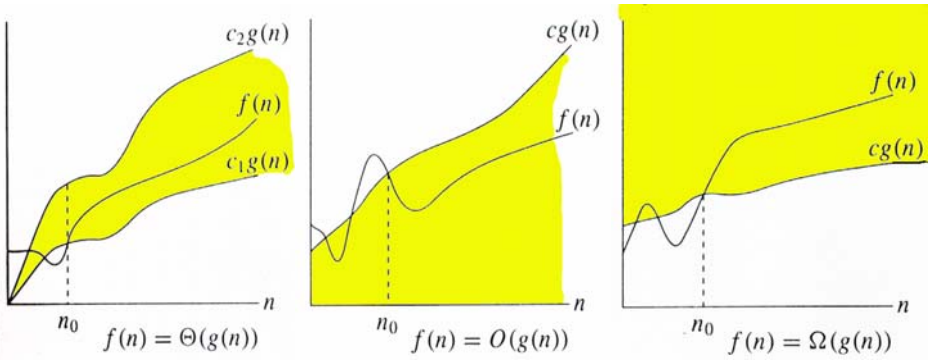
Asymptotic Complexity

- Running time of an algorithm as a function of input size n **for large** ***n***.
- Expressed using only the **highest-order term** in the expression for the exact running time.
  - Instead of exact running time, say $\Theta(n^2)$.
- Describes behavior of function in the limit.
- Written using ***Asymptotic Notation***.

Asymptotic Notation

- **Θ, *O*, Ω, *o*, ω**
- Defined for functions over the natural numbers.
  - **Ex:** $f(n) = \Theta(n^2)$.
  - Describes how $f(n)$ grows in comparison to n^2.
- Define a ***set*** of functions; in practice used to compare two function sizes.
- The notations describe different rate-of-growth relations between the defining function and the defined set of functions.

Relations Between Θ, *O*, Ω



Agenda

1. Introduction

2. Graphs

3. **Preview of Algorithm Designs and Analysis**

4. Divide and Conquer

5. Dynamic Programming

6. Greedy Methods

7. NP Completeness

Preview of Algorithm Designs and Analysis

- Incremental Approach
- Divide and Conquer
- Randomization (Randomized Algorithm)
- Linear Programming
- Dynamic Programming
- Greedy method
- Amortized analysis
- Brute-force
- Backtracking
- Branch and Bound

Divide and Conquer

- Recall the key idea of Divide and Conquer
- Recursive in structure
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - ***Combine*** the solutions to create a solution to the original problem

Dynamic Programming

- ***Dynamic programming*** (DP) is typically applied to optimization problems. In such problem there can be ***many solutions***. Each solution has a value, and we wish to find *a* ***solution*** with the optimal value.
- Example problems: 0/1 knapsack problem, Matrix Multiplication Chains problem
- **Like** divide and conquer, DP solves problems by combining solutions to subproblems.

Dynamic Programming

- **Unlike** divide and conquer, subproblems are not independent.
  - Subproblems may share subsubproblems,
  - However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.)
- DP reduces computation by
  - Solving subproblems in a bottom-up fashion.
  - Storing solution to a subproblem the first time it is solved.
  - Looking up the solution when subproblem is encountered again.
- Key: determine structure of optimal solutions

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Dynamic Programming

- The development of a dynamic programming algorithm can be broken into a sequence of four steps:
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution in a bottom up fashion.
  4. Construct an optimal solution from computed information.

Greedy Method

- Like dynamic programming, used to solve optimization problems.
- Problems exhibit optimal substructure (like DP).
- Problems also exhibit the **greedy-choice** property.
  - When we have a choice to make, make the one that looks best *right now*.
  - Make a **locally optimal choice** in hope of getting a **globally optimal solution**.

Greedy Method

- The choice that seems best at the moment is the one we go with.
  - Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
  - Show that all but one of the subproblems resulting from the greedy choice are empty.
- Example Problems: Container Loading problem, 0/1 Knapsack problem

Brute-Force

- *Brute-force algorithms* are distinguished not by their structure or form, but by the way in which the problem to be solved is approached.
- A brute-force algorithm solves a problem in the most simple, direct or obvious way. As a result, such an algorithm can end up doing far more work to solve a given problem than a more clever or sophisticated algorithm might do.
- On the other hand, a brute-force algorithm is often easier to implement than a more sophisticated one and, because of this simplicity, sometimes it can be more efficient.

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Brute-Force

- Often a problem can be viewed as a sequence of decisions to be made.
  - For example, consider the problem of finding the best way to place electronic components on a circuit board.
  - To solve this problem we must decide where on the board to place each component.
- Typically, a brute-force algorithm solves such a problem by exhaustively enumerating all the possibilities. I.e., for every decision we consider each possible outcome.
- Example Problems: 0/1 knapsack problem, Sequential Search problem, Hamilton Circuits problem, Board Permutation problem

Backtracking

- *Backtracking* is a refinement of the brute force approach, which systematically searches for a solution to a problem among all available options.
- A *backtracking* algorithm systematically considers all possible outcomes for each decision.
- In this sense, backtracking algorithms are like the brute-force algorithms.
- However, backtracking algorithms are distinguished by the way in which the space of possible solutions is explored.
- Sometimes a backtracking algorithm can detect that an exhaustive search is unnecessary and, therefore, it can perform much better.

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Backtracking

- It does so by assuming that the solutions are represented by vectors (v_1, ..., v_m) of values and by traversing, in a depth first manner, the domains of the vectors until the solutions are found.
- When invoked, the algorithm starts with an empty vector. At each stage it extends the partial vector with a new value.
- Upon reaching a partial vector (v_1, ..., v_i) which can't represent a partial solution, the algorithm backtracks by removing the trailing value from the vector, and then proceeds by trying to extend the vector with alternative values.
- Example Problems: 0/1 knapsack problem, Container Loading problem, Max Clique problem, Traveling Salesman problem, Board Permutation problem

Branch and Bound

- *Branch-and-bound* is an approach developed for solving discrete and combinatorial optimization problems.
- The discrete optimization problems are problems in which the decision variables assume discrete values from a specified set.
- The combinatorial optimization problems, on the other hand, are problems of choosing the best combination out of all possible combinations.
- Example problems: 0/1 knapsack problem, Container Loading problem, Max Clique problem, Traveling Salesman problem, Board Permutation problem

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Branch and Bound

- The essence of the branch-and-bound approach is the following observation:
  - In the total enumeration tree, at any node, if I can show that the optimal solution cannot occur in any of its descendents, then there is no need for me to consider those descendent nodes.
  - Hence, I can ***"prune"*** the tree at that node. If I can prune enough branches of the tree in this way, I may be able to reduce it to a computationally manageable size.
  - Note that, I am *not* ignoring those solutions in the *leaves* of the branches that I have pruned, I have left them out of consideration *after* I have made sure that the optimal solution cannot be at any one of these nodes.
- Thus, the branch-and-bound approach is not a heuristic, or approximating, procedure, but it is an exact, optimizing procedure that finds an optimal solution.

延伸閱讀

- http://en.wikipedia.org/wiki/Algorithm_design
- http://www.csie.ntnu.edu.tw/~u91029/AlgorithmDesign.html
- http://code.tutsplus.com/tutorials/understanding-the-principles-of-algorithm-design--net-26561
- https://www.youtube.com/watch?v=a_otxyu0mSQ
- https://www.youtube.com/watch?v=Qe6PUzVu2pk
- https://www.youtube.com/watch?v=SDgo4kVSiiw

Agenda

1. Introduction
2. Graphs
3. Preview of Algorithm Designs and Analysis
4. **Divide and Conquer**
5. Dynamic Programming
6. Greedy Methods
7. NP Completeness

Divide and Conquer

- Famous Problems and Algorithms
  - Merge Sort
  - Quick Sort
  - Binary Search
  - Binary Tree traversals
  - Multiplication of Large Integers
  - Matrix Multiplication
  - Closest Pair of Points
  - Convex-Hull

Example Problem: Closest Pair

- Define Problem
  - **Input:**

    $P = \{p(1), p(2), ..., p(n)\}$ where $p(i) = (x(i), y(i))$.

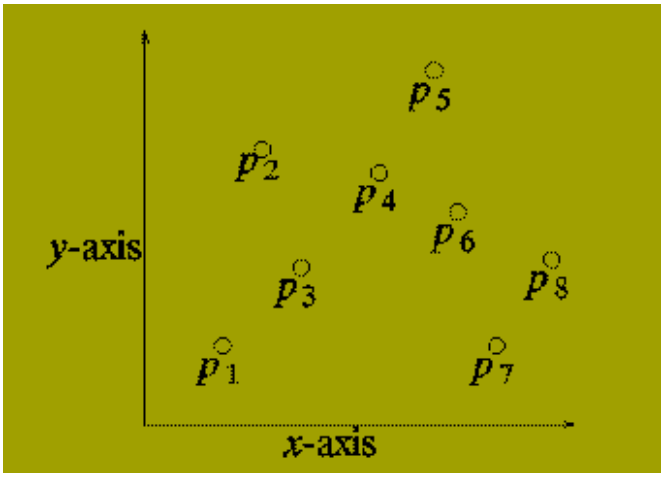
    A set of n points in the plane.
  - **Output**

    The *distance* between the two points that are closest.

Note: The distance $DELTA(i, j)$ between $p(i)$ and $p(j)$ is defined by the expression:

Square root of { (x(i)-x(j))^2 + (y(i)-y(j))^2 }

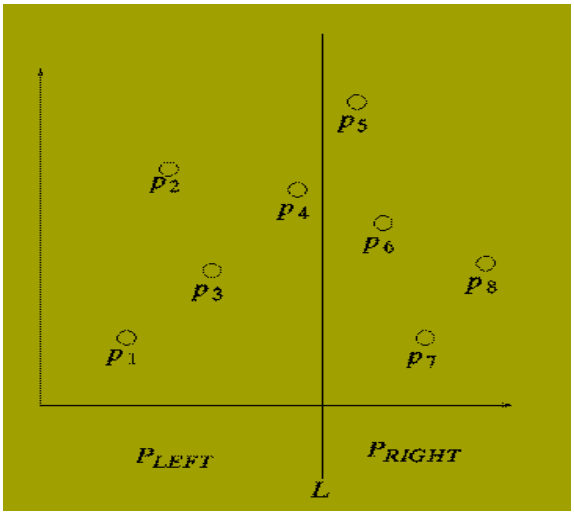
Closest Pair Problem



Solved in Divide and Conquer(1/3)

- We assume that:
- *n* is an exact power of 2, *n = 2^k*.
- For each *i*, *x(i) < = x(i+1)*, i.e. the points are ordered by increasing *x* from left to right.
- Consider drawing a vertical line (*L*) through the set of points *P* so that half of the points in *P* lie to the *left* of *L* and half lie to the *right* of *L*.

Solved in Divide and Conquer(2/3)



Solved in Divide and Conquer(3/3)

- There are three possibilities:
  - The closest pair lie in *P-LEFT*.
  - The closest pair lie in *P-RIGHT*.
  - The closest pair contains:

    One Point from *P-LEFT*

    and

    One Point from *P-RIGHT*
- So we have a (rough) Divide-and-Conquer Method as follows:

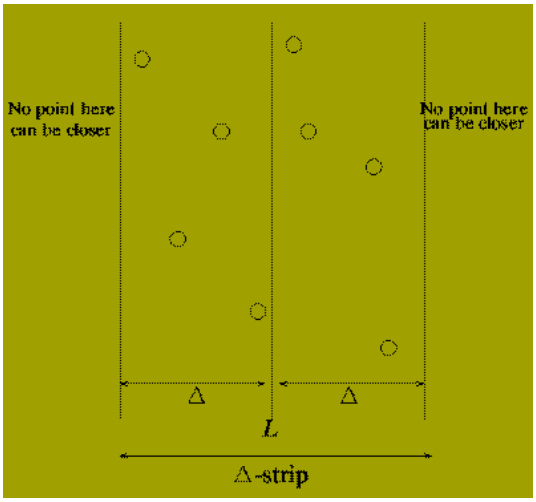
Algorithm for Closest Pair Problem

```
function closest_pair (P: point set; n: integer )
  float DELTA-LEFT, DELTA-RIGHT : float; DELTA :
    begin
      if n = 2 then return distance from p(1) to p(2);
      else P-LEFT := ( p(1), p(2) ,..., p(n/2) );
           P-RIGHT := ( p(n/2+1), p(n/2+2) ,..., p(n) );
          DELTA-LEFT := closest_pair( P-LEFT, n/2 );
         DELTA-RIGHT := closest_pair( P-RIGHT, n/2 );
        DELTA := minimum ( DELTA-LEFT, DELTA-RIGHT );
--*********************************************
Determine whether there are points p(l) in P-LEFT and p(r) in P-RIGHT with distance( p(l), p(r) ) < DELTA.
 If there are such points, set DELTA to be the smallest distance.
--*********************************************
          return DELTA; end if;
end closest_pair;
```


Combine (1/3)

- The section between the two comment lines is the `combine' stage of the Divide-and-Conquer algorithm.
- If there are points *p(l)* and *p(r)* whose distance apart is less than *DELTA* then it must be the case that
  - The *x*-coordinates of *p(l)* and *p(r)* differ by at most *DELTA*.
  - The *y*-coordinates of *p(l)* and *p(r)* differ by at most *DELTA*.

Combine (2/3)



Combine (3/3)

- The combine stage can be implemented by:
  - Finding all points in *P-LEFT* whose *x*-coordinate is at least *x(n/2)-DELTA*.
  - Finding all points in *P-RIGHT* whose *x*-coordinate is at most *x(n/2)+DELTA*.
- Call the set of points found in (1) and (2) *P-strip*. and sort the *s* points in this in order of increasing *y*-coordinate. letting *(q(1),q(2) ,..., q(s))* denote the sorted set of points.
- Then the combine stage of the algorithm consists of two nested **for** loops:

Statements of Combine section

```
for i in 1..s loop
   for j in i+1..s loop
      exit when (| x(i) - x(j) | > DELTA or | y(i) - y(j) | > DELTA);
         if distance( q(i), q(j) ) < DELTA then DELTA := distance ( q(i), q(j) );
         end if;
   end loop;
end loop;
```


延伸閱讀

- http://en.wikipedia.org/wiki/Divide_and_conquer_algorithms
- http://acm.nudt.edu.cn/~twcourse/DivideAndConquer.html
- https://www.cs.berkeley.edu/~vazirani/algorithms/chap2.pdf
- http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/divide.htm
- https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms
- https://www.youtube.com/watch?v=pJBG5_ja_YU
- https://www.youtube.com/watch?v=ATCYn9F3oUQ
- https://www.youtube.com/watch?v=6SUmp_Cn-SU

Agenda

1. Introduction

2. Graphs

3. Preview of Algorithm Designs and Analysis

4. Divide and Conquer

5. Dynamic Programming

6. Greedy Methods

7. NP Completeness

Dynamic Programming

- Famous Problems and Algorithms
  - Matrix-Chain Multiplication
  - Longest Common Subsequence
  - Optimal Binary Search Trees
  - The knapsack Problem
  - Single-Source Shortest Path
  - All Pairs Shortest Paths
  - Image compression
  - Component Folding
  - Noncrossing Subset of Nets
  - Computing a Binomial Coefficient

Dynamic Programming

- ***Dynamic programming*** is typically applied to optimization problems. In such problem there can be ***many solutions***. Each solution has a value, and we wish to find *a* ***solution*** with the optimal value.

Matrix-chain multiplication (1/2)

- A product of matrices is **fully parenthesized** if it is either a single matrix, or a product of two fully parenthesized matrix product, surrounded by parentheses.

Matrix-chain multiplication (2/2)

- How to compute $A_1 A_2 \ldots A_n$ where A_i is a matrix for every i.
- Example: $A_1 A_2 A_3 A_4$

$(A_1(A_2(A_3 A_4)))$ $(A_1((A_2 A_3)A_4))$
$((A_1 A_2)(A_3 A_4))$ $((A_1(A_2 A_3))A_4)$
$(((A_1 A_2)A_3)A_4)$

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

MATRIX MULTIPLY

MATRIX MULTIPLY(A,B)

1 **if** columns[A] $\neq$ column[B]
2 **then error** "incompatible dimensions"
3 **else for** $i \leftarrow 1$ **to** rows[A]
4 **do for** $j \leftarrow 1$ **to** columns[B]
5 **do** $c[i,j] \leftarrow 0$
6 **for** $k \leftarrow 1$ **to** columns[A]
7 **do** $c[i,j] \leftarrow c[i,j] + A[i,k]B[k,j]$
8 **return** C

Complexity:

- Let A be a $p \times q$ matrix, and B be a

$q \times r$ matrix. Then the complexity is

$p \times q \times r$

Example:

- A_1 is a 10×100 matrix, A_2 is a 100×5 matrix, and A_3 is a 5×50 matrix. Then

$((A_1 A_2) A_3)$ takes $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ time. However, $(A_1 (A_2 A_3))$ takes

$100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$ time.

The matrix-chain multiplication problem

- Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i=0,1,\ldots,n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

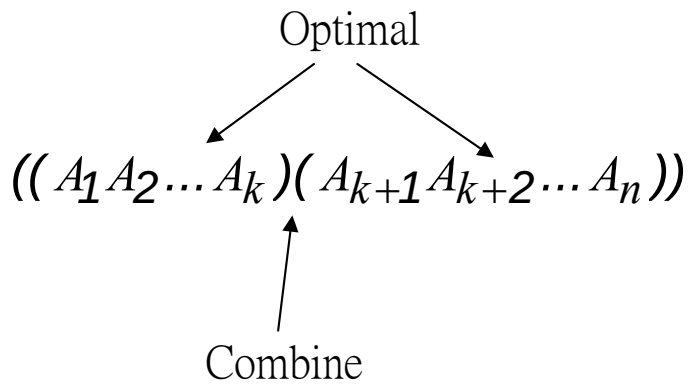
Counting the number of parenthesizations

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)p(n-k) & \text{if } n \geq 2 \end{cases}$$

- $P(n) = C(n-1)$ [Catalan number]

$$= \frac{1}{n+1}\binom{2n}{n} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

Step 1: The structure of an optimal parenthesization



Step 2: A recursive solution

- Define $m[i, j]$= minimum number of scalar multiplications needed to compute the matrix $A_{i..j} = A_iA_{i+1}..A_j$
- goal $m[1, n]$

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & i \ne j \end{cases}$$

Step 3: Computing the optimal costs

Using MATRIX_CHAIN_ORDER algorithm

MATRIX_CHAIN_ORDER

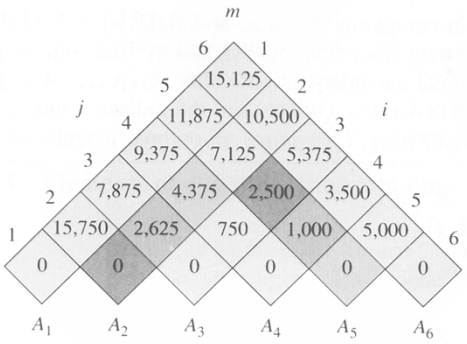
```
MATRIX_CHAIN_ORDER(p)
1   n ← length[p] –1
2   for i ← 1 to n
3         do m[i, i] ← 0
4   for l ← 2 to n
5         do for   i ← 1 to n – l + 1
6                  do j ← i + l – 1
7                     m[i, j] ← ∞
8                     for k ← i to j – l
9                           do q ← m[i, k] + m[k+1, j]+ p_{i-1} p_k p_j
10                             if q < m[i, j]
11                                   then m[i, j] ← q
12                                        s[i, j] ← k
13  return m and s
```

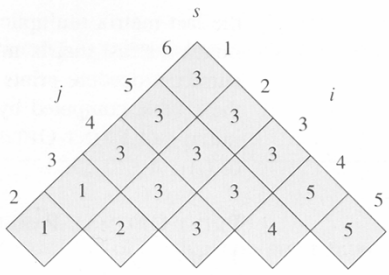
Complexity: $O(n^3)$

Example

$$
\begin{aligned}
A_1 \quad & 30 \times 35 \quad = p_0 \times p_1 \\
A_2 \quad & 35 \times 15 \quad = p_1 \times p_2 \\
A_3 \quad & 15 \times 5 \quad = p_2 \times p_3 \\
A_4 \quad & 5 \times 10 \quad = p_3 \times p_4 \\
A_5 \quad & 10 \times 20 \quad = p_4 \times p_5 \\
A_6 \quad & 20 \times 25 \quad = p_5 \times p_6
\end{aligned}
$$

The ***m*** and ***s*** table computed by MATRIX-CHAIN-ORDER for n=6





Result

$m[2,5]=$

min{

$m[2,2]+m[3,5]+p_1p_2p_5=0+2500+35\times15\times20=13000,$

$m[2,3]+m[4,5]+p_1p_3p_5=2625+1000+35\times5\times20=7125,$

$m[2,4]+m[5,5]+p_1p_4p_5=4375+0+35\times10\times20=11374$

}

=7125

Step 4: Constructing an optimal solution

Using MATRIX_CHAIN_MULTIPLY algorithm

MATRIX_CHAIN_MULTIPLY

MATRIX_CHAIN_MULTIPLY(A, s, i, j)

1 **if** $j > i$

2 **then** $x \leftarrow MCM(A, s, i, s[i, j])$

3 $y \leftarrow MCM(A, s, s[i, j] + 1, j)$

4 **return** MATRIX-MULTIPLY(X,Y)

5 **else return** A_i

Result of the Example

$$((A_1(A_2A_3))((A_4A_5)A_6))$$

Elements of dynamic programming

- **Optimal substructure**
- **Subtleties**
- **Overlapping subproblems**

Optimal substructure (1/3)

- We say that a problem exhibits **optimal substructure** if an optimal solution to the problem contains within its optimal solution to subproblems.
- Example: Matrix-multiplication problem

Optimal substructure (2/3)

1. You show that a solution to the problem consists of making a choice, Making this choice leaves one or more subproblems to be solved.
2. You suppose that for a given problem, you are given the choice that leads to an optimal solution.
3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
4. You show that the solutions to the subproblems used within the optimal solution to the problem must themselves be optimal by using a "***cut-and-paste***" technique.

Optimal substructure (3/3)

Optimal substructure varies across problem domains in two ways:

1. how many subproblems are used in an optimal solutiion to the original problem, and
2. how many choices we have in determining which subproblem(s) to use in an optimal solution.

Subtleties

- One should be careful not to assume that optimal substructure applies when it does not. consider the following two problems in which we are given a directed graph $G = (V, E)$ and vertices $u, v \in V$.
  - Unweighted shortest path:
    - Find a path from u to v consisting of the fewest edges. Good for Dynamic programming.
  - Unweighted longest simple path:
    - Find a simple path from u to v consisting of the most edges. Not good for Dynamic programming.

Overlapping Subprogrammings

- example: MAXTRIX_CHAIN_ORDER

Longest Common Subsequence

- **Longest Common Subsequence** is the problem of finding the longest common subsequence of two sequences of items. This is used in the "diff" file comparison utility.
- Given two sequence of items, find the longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, in the string *abcdefg*, "abc", "abg", "bdf", "aeg" are all subsequences.
- A naive exponential algorithm is to notice that a string of length n has $O(2^n)$ different subsequences, so we can take the shorter string, and test each of its subsequences for presence in the other string, greedily.

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Longest Common Subsequence

- **Recursive solution**
- We can try to solve the problem in terms of smaller subproblems. We are given two strings x and y, of length n and m respectively. We solve the problem of finding the longest common subsequence of $x = x_{1...n}$ and $y = y_{1...m}$ by taking the best of the three possible cases:
  - The longest common subsequence of the strings $x_{1...n-1}$ and $y_{1...m}$
  - The longest common subsequence of the strings $x_{1...n}$ and $y_{1...m-1}$
  - If x_n is the same as y_m, the longest common subsequence of the strings $x_{1...n-1}$ and $y_{1...m-1}$, followed by the common last character.
- It is easy to construct a recursive solution from this:

Longest Common Subsequence

```
func lcs(x,y)
    if ( length(x)=0 or length(y)=0 )
        return ""
    best = lcs(x[1,n-1],y[1,m])
    if ( length(best) < length(lcs(x[1,n],y[1,m-1])) )
        best = lcs(x[1,n],y[1,m-1])
    if ( x[n] = y[m] and length(best) < length(lcs(x[1,n-1],y[1,m-1]) 1 )
        best = lcs(x[1,n-1],y[1,m-1]) x[n]
    return best
```


TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Longest Common Subsequence

- **Dynamic programming**
- Obviously, this is still not very efficient. But because the subproblems are repeated, we can use memoization. An even more (slightly) efficient way, which avoids the overhead of function calls, is to order the computation in such a way that whenever the results of subproblems are needed, they have already been computed, and can simply be looked up in a table. This is called Dynamic Programming.
- In this case, we find lcs($x_{1..i}$, $y_{1..j}$) for every i and j, starting from smaller ones, storing the results in an array at index (i,j) as we go along

<inhon@mail.tku.edu.tw> May 31, 2015

Longest Common Subsequence

```
LCS − Length(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4        do c[i, 0] ← 0
 5  for j ← 0 to n
 6        do c[0, j] ← 0
 7  for i ← 1 to m
 8        do for j ← 1 to n
 9              do if x_i = y_j
10                    then c[i, j] ← c[i − 1, j − 1] + 1
11                         b[i, j] ← "↖"
12                    else if c[i − 1, j] ≥ c[i, j − 1]
13                            then c[i, j] ← c[i − 1, j]
14                                 b[i, j] ← "↑"
15                            else c[i, j] ← c[i, j − 1]
16                                 b[i, j] ← "←"
17  return c and b
```


Longest Common Subsequence

```
func lcs(x,y)
    n = length( x ), m = length( y )
    for i from 0 to n
        for j from 0 to m
            if ( i is 0 or j is 0 )
                table[i,j] = ""
                if ( x[i] == y[j] ) table[i,j] = x[i]
            else /* Sentinel */
                table[i,j] = table[i-1,j]
                if ( length( table[i,j] ) < length( table[i,j-1] ) ) table[i,j] = table[i,j-1];
                if ( x[i] = y[j] and length( table[i,j] ) < length( table[i-1,j-1] ) ) table[i,j] = table[i-1,j-1];
        return table[n][m]
```

Longest Common Subsequence

- Example
  - Two sequences as follows
    - HUMAN
    - CHIMPANZE
  - The LCS of the two sequences is
    - HMAN
      - HUMAN
      - CHIMPANZE

0/1 Knapsack Problem

- You have a knapsack that has capacity (weight) C.
- You have several items J_1, …, J_n.
- Each item J_i has a weight w_i and a benefit p_i.
- You want to place a certain number of item J_i in the knapsack so that:
  - The knapsack weight capacity is not exceeded and
  - The total benefit is maximal.

Dynamic Programming solves 0/1 Knapsack Problem

- The structure of Optimal solution
  - maximize $\sum_{j=1}^{n} p_j x_j$
  - subject to $\sum_{j=1}^{n} w_j x_j \le W, \quad x_j \in \{0,1\}$
- The recursive subtleties
  - $M(n, C) = \max \{M(n-1, C), \quad \text{if } n>0 \ \& \ C-w_i < 0$

    $M(n-1, C-w_i) + p_i, \quad \text{if } n>0 \ \& \ C-w_i \ge 0$

    $0, \quad \text{if } n = 0 \ \& \ C > 0 \ \}$

Example

Item	Weight	Benefit
A	2	60
B	3	75
C	4	90

Capacity = 5

Example

- Suppose f(*C*) represents the *maximal possible benefit* of a knapsack with Capacity *C*.
- We want to find f(5).
- Recursive Calculation
  - f(C) = MAX { p_j + f(C-w_j) | I_j is an item}.

Example

- f(0), f(1)
- f(0) = 0. Why? The knapsack with capacity 0 can have nothing in it.
- f(1) = 0. There is no item with weight 1.

Example

- f(2)
- f(2) = 60. There is only one item with Benefit 60.
- **Choose A.**

Example

- f(3)
- f(3) = MAX { b_j + f(w-w_j) | I_j is an item}.

= MAX { 60+f(3-2), 75 + f(3-3)}

= MAX { 60 + 0, 75 + 0 }

= 75.

- **Choose B.**

Example

- f(4)
- f(4) = MAX { b_j + f(w-w_j) | I_j is an item}.

= MAX { 60 + f(4-2), 75 + f(4-3), 90+f(4-4)}

= MAX { 60 + f(2), 75 + f(1), 90 + f(0)}

= MAX { 60 + 0, 75 + 0, 90 + 0}

=90.

- **Choose C.**

Example

- f(5)
- f(5) = MAX { b_j + f(w-w_j) | I_j is an item}.

= MAX { 60 + f(5-2), 75 + f(5-3), 90+f(5-4)}

= MAX { 60 + f(3), 75 + f(2), 90 + f(1)}

= MAX { 60 + 75, 75 + 60, 90 + 0}

=135.

- **Choose A + B.**

Example

- Optimal knapsack Benefit is 135
- Remain capacity = 0.
- The optimal solutions: Take A and B

Another Example

Item	Weight	Benefit
A	4Kg	$4500
B	5Kg	$5700
C	2Kg	$2250
D	1Kg	$1100
E	6Kg	$6700

Capacity = 8 Kg

延伸閱讀

- http://en.wikipedia.org/wiki/Dynamic_programming
- http://www.csie.ntnu.edu.tw/~u91029/DynamicProgramming.html
- http://acm.nudt.edu.cn/~twcourse/DynamicProgramming.html
- https://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf
- https://www.youtube.com/watch?v=OQ5jsbhAv_M
- https://www.youtube.com/watch?v=gm9QkcdIN9o
- https://www.youtube.com/watch?v=sF7hzgUW5uY
- https://www.youtube.com/watch?v=PLJHuErj-Tw
- https://www.youtube.com/watch?v=UhFvK3uERGg

Agenda

1. Introduction
2. Graphs
3. Preview of Algorithm Designs and Analysis
4. Divide and Conquer
5. Dynamic Programming
6. **Greedy Methods**
7. NP Completeness

Greedy Methods

- Famous Problems and Algorithms
  - Minimum Cost Spanning Tree
  - Single-Source Shortest Path
  - Bipartite Cover
  - Topological Sorting
  - The Knapsack problem
  - Container Loading
  - Task Scheduling
  - Huffman Code

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Greedy Methods

- Like dynamic programming, used to solve optimization problems.
- Problems exhibit optimal substructure (like DP).
- Problems also exhibit the **greedy-choice** property.
  - When we have a choice to make, make the one that looks best *right now*.
  - Make a **locally optimal choice** in hope of getting a **globally optimal solution**.

<inhon@mail.tku.edu.tw> May 31, 2015

Greedy Methods

- Greedy Strategy
  - The choice that seems best at the moment is the one we go with.
    - Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
    - Show that all but one of the subproblems resulting from the greedy choice are empty.

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Greedy Methods

- Elements of Greedy Algorithms
  - Greedy-choice Property.
    - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
  - Optimal Substructure.

Greedy Methods

- Examples
  - Minimal Cost Spanning Tree
  - Knapsack Problem (0/1, Fractional, Multiple items)
  - Huffman Codes

Recall 0/1 Knapsack Problem

- You have a knapsack that has capacity (weight) C.
- You have several items $I_1,\ldots,I_n$.
- Each item I_j has a weight w_j and a benefit b_j.
- You want to place a certain number of item I_j in the knapsack so that:
  - The knapsack weight capacity is not exceeded and
  - The total benefit is maximal.

0/1 Knapsack Problem

- Two kinds of selected directions
  - Best Benefit only
  - Best Unit_Benefit (Cost)
- $f(C) = f(C - w_j) + f(C - w_j - w_{j+1}) + \ldots + f(0)$ or no more item can be added
  - which I_j is the Best Choice, I_{j+1} is the second Best Choice, and so on.

Recall Example

Item	Weight	Benefit
A	2	60
B	3	75
C	4	90

Capacity = 5

Choice by Benefit only

- Step 1. Choose the item with Max. Benefit,
  - Choose Item C, Benefit = 90, remain Capacity = 1
- Step 2. No more item with weight less then remain Capacity, Stop
- Take Item C, Total Benefit is 90

Choice by Cost (Benefit / Weight)

- Step 1. Choose the item with Max. Cost,
  - Item A, Cost = 30
  - Item B, Cost = 25
  - Item C, Cost = 22.5
  - Choose Item A, remain Capacity = 3
- Step 2. Choose the next item with Max. Cost,
  - Choose Item B, remain Capacity = 0
- Remain Capacity = 0, Stop
- Take Items A and B, Total Benefit is 135

Another Example

- Knapsack of capacity 50.
- 3 items
  - Item 1 has weight 10, benefit 60
  - Item 2 has weight 20,benefit 100
  - Item 3 has weight 30, benefit 120
- Apply two kinds of selection criteria to find the solution by Greedy Method
  - Benefit only: take Items 3 and 2, Total benefit is 220
  - Cost: take Item 1 and 2, Total benefit is 160

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Another Example

Item	Weight	Benefit
A	4Kg	$4500
B	5Kg	$5700
C	2Kg	$2250
D	1Kg	$1100
E	6Kg	$6700

Capacity = 8 Kg

Huffman Coding

- **Huffman coding** is an entropy encoding algorithm used for lossless data compression.
- The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol.
  - It was developed by David A. Huffman while he was a Ph.D. student at MIT

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Huffman Coding

- Main properties：
  - Use variable-length code for encoding a source symbol.
  - Shorter codes are assigned to the most frequently used symbols, and longer codes to the symbols which appear less frequently.
  - Unique decodable & Instantaneous code.
  - It was shown that Huffman coding cannot be improved or with any other integral bit-width coding stream.

Huffman Coding

- Compare to ASCII (Examples), *fixed-length* codes

Character	Binary Code	Hexadecimal Code
A	01000001	41
J	01001010	4A
V	01010110	56
#	00100011	23
a	01100001	61
n	01101110	6E
t	01110100	74
~	01111110	7E

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Huffman Coding

- Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix code (sometimes called "prefix-free codes") that expresses the most common characters using shorter strings of bits than are used for less common source symbols.
- Huffman was able to design the most efficient compression method *of this type*: no other mapping of individual source symbols to unique strings of bits will produce a smaller average output size when the actual symbol frequencies agree with those used to create the code.
- A method was later found to do this in linear time if input probabilities (also known as *weights*) are sorted.

Huffman Tree

- Definition
  - A **Huffman tree** is a binary tree which minimizes the sum of f(i)D(i) over all leaves i,
    - where f(i) is the frequency or weight of leaf i, and
    - D(i) is the length of the path from the root to leaf i.
    - In each of the applications, f(i) has a different physical meaning.
- Properties
  - Every internal node has 2 children.
  - Smaller frequencies are further away from the root.
  - The 2 smallest frequencies are siblings.

Example of Huffman Tree and Coding

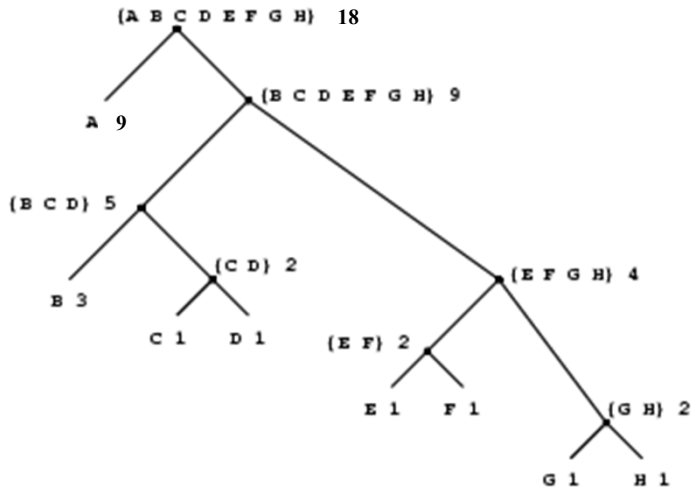
- If all our messages are made up of the eight symbols A, B, C, D, E, F, G, and H
- We can choose a code with three bits per character
  - For example A 000, B 001, C 010, D 011, E 100, F 101
  - G 110, H 111
- With this code, the message
  - BACADAEAFABBAAAGAH
- is encoded as the string of 54 bits
  - 001000010000011000100000101000001001000000000110000111

Example of Huffman Tree and Coding

- We count the frequency of each character shown in the message.

Character	Frequency
A	9
B	3
C	1
D	1
E	1
F	1
G	1
H	1

Example of Huffman Tree and Coding



Example of Huffman Tree and Coding

- Thus, we encode the character with *variable-lengh codes as follow.*

A 0, B 100, C 1010, D 1011, E 1100, F 1101 , G 1110, H 1111

Character	Frequency	**Huffman Codes**
A	9	0
B	3	100
C	1	1010
D	1	1011
E	1	1100
F	1	1101
G	1	1110
H	1	1111

Example of Huffman Tree and Coding

- With Huffman coding, the message
  - BACADAEAFABBAAAGAH
- is encoded as the string of 42 bits
  - 100010100101101100011010100100000111001111

Another Example

Character	Frequency	Huffman Codes
Space	7	
a	4	
e	4	
f	3	
h	2	
i	2	
l	1	
m	2	
n	2	
o	1	
p	1	
r	1	
s	2	
t	2	
u	1	
x	1	

延伸閱讀

- http://en.wikipedia.org/wiki/Greedy_algorithm
- http://ccckmit.wikidot.com/so:greedyalgorithm
- http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Greedy/greedyIntro.htm
- https://www.youtube.com/watch?v=A8CEvPmNpKQ
- https://www.youtube.com/watch?v=ZFK9_jgCBrE
- htps://www.youtube.com/watch?v=apcCVfXfcqE

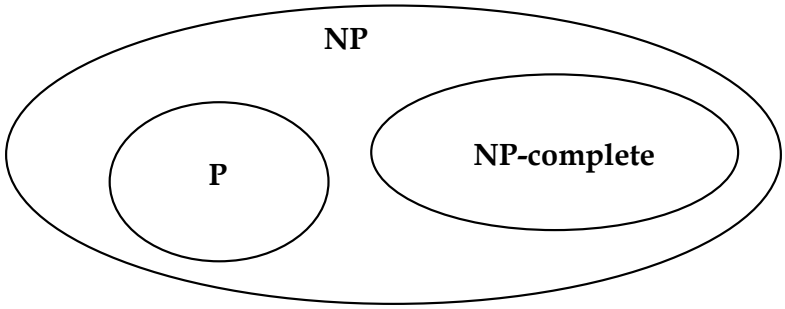
Agenda

1. Introduction
2. Graph
3. Preview of Algorithm Designs and Analysis
4. Divide and Conquer
5. Dynamic Programming
6. Greedy Methods
7. **NP Completeness**

Recall NP Problem

- A problem is assigned to the NP (nondeterministic polynomial time) class if it cannot be solved in polynomial time.
- A problem is said to be NP-hard if an algorithm for solving it can be translated into one for solving any other NP-problem. It is much easier to show that a problem is NP than to show that it is NP-hard.
- A problem which is both NP and NP-hard is called an NP-complete problem.

Recall NP Problem



NP Completeness

- Famous Problems
  - Hamiltonian cycle in a directed graph.
  - 3-CNF satisfiability.
  - Circuit satisfiability problem.
  - Longest simple paths in a directed graph.
  - Vertex-cover problem in an undirected graph.
  - Traveling-salesman problem.

NP Completeness

- The problems we are trying to solve are basically of two kinds.
- In *decision problems* we are trying to decide whether a statement is true or false.

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

NP Completeness

- In *optimization problems* we are trying to find the solution with the best possible score according to some scoring scheme.
- Optimization problems can be either *maximization problems*, where we are trying to maximize a certain score, or *minimization problems*, where we are trying to minimize a cost function.

Hamiltonian cycles

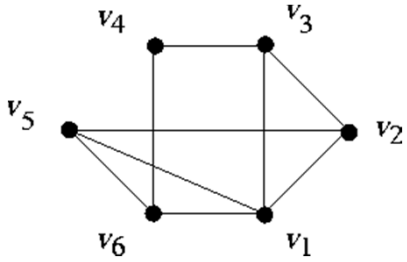
- Given a directed graph, we want to decide whether or not there is a Hamiltonian cycle in this graph.
- This is a decision problem.

Traveling-salesman problem

- Given a complete graph and an assignment of weights to the edges, find a Hamiltonian cycle of minimum weight.
- This is the *optimization version* of the problem. In the *decision version*, we are given a weighted complete graph and a real number *c,* and we want to know whether or not there exists a Hamiltonian cycle whose combined weight of edges does not exceed *c.*

The Hamiltonian Cycle Problem

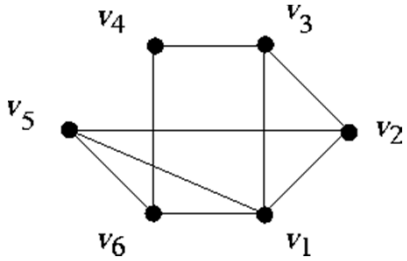
- Let ***G*** be a finite graph with ***V*(*G*)** the set of vertices and ***E*(*G*)** the set of edges. A ***Hamiltonian cycle c*** of ***G*** is a cycle that goes through every vertex exactly once. The ***Hamiltonian cycle problem (HCP)*** asks whether a given graph ***G*** has a Hamiltonian cycle



TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

The Hamiltonian Cycle Problem

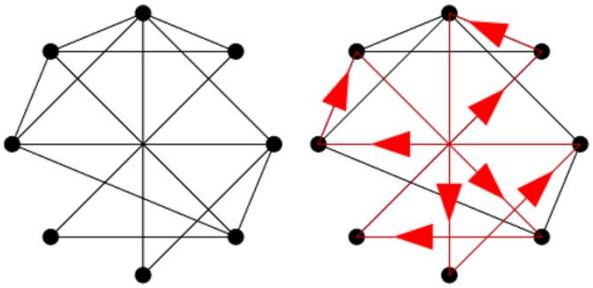
- The cycle *v*1*v*6*v*4*v*3*v*2*v*5*v*1 is a Hamiltonian cycle. Of course, there are many other cycles that are not Hamiltonian, for example, *v*1*v*6*v*5*v*1 or the loop *v*2*v*5*v*1*v*2*v*5*v*1*v*2



<inhon@mail.tku.edu.tw> May 31, 2015

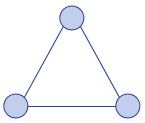
Traveling-salesman problem

- The traveling salesman problem (TSP) asks for the shortest route to visit a collection of cities and return to the starting point. Despite an intensive study by mathematicians, computer scientists, operations researchers, and others, over the past 50 years, it remains an open question whether or not an efficient general solution method exists



Traveling-salesman problem

- 若採用暴力法去解TSP問題，則會發現要找出所有可能的路徑所花費的時間是呈**指數（Exponentially）**成長的!!
  - 3 cities → 1 solution.
  - 10 cities → 181,440 possible tours
  - n cities → (n-1)!/2 possible tours



Traveling-salesman problem

- 若 n=26，則有 25! /2條不同路徑：
  - 25!=15511210043330985984000000≅1.55 x 10^{25}這個數字寫來輕鬆，究竟有多大？
  - 假設電腦每秒可計算 10^6 條路徑的成本，一年有 3.15 x 10^7秒， 故一年可計算 3.15 x 10^{13}條路徑，求出所有路徑的成本需時

Traveling-salesman problem

$$\frac{1.55 \times 10^{25}}{3.15 \times 10^{13}} \cong 5 \times 10^{11}(\text{年})$$

- 即便是對不太大的 n=26，就需時五千億年，顯然這種方法毫無用處。

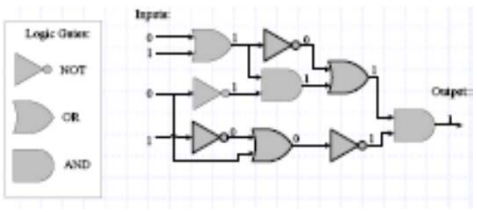
CIRCUIT-SAT Problem

- A Boolean circuit is a circuit of AND, OR, and NOT gates; the CIRCUIT-SAT problem is to determine if there is an assignment of 0's and 1's to a circuit's inputs so that the circuit outputs 1.
- The Circuit-SAT Problem is a NP-Complete Problem

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

CIRCUIT-SAT Problem

- Non-deterministically choose a set of inputs and the outcome of every gate, then test each gate's I/O.



P vs. NP

- In each of the following pairs of problems, one is solvable in polynomial time and the other is NPC (NP-Completeness).

1. **Shortest path vs. longest simple path**
   a. **Shortest path** (even with negative edge weights) is solvable in **polynomial time**. We can find a shortest path from a single source to a single destination in a graph, $G = (V,E)$, in $O(VE)$ time.
   b. Finding the **longest simple path** between two vertices in $G = (V,E)$ is **NPC**. "Simple" means the path does not cross over itself by going the same vertex more than once.

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

P vs. NP

2. **Euler Tour vs. Hamiltonian Cycle**
   - a. An **Euler Tour** traverses each edge of a directed graph $G = (V,E)$ exactly once, although it may visit a vertex more than once. If a graph has an Euler Tour we can find the edges in $\boldsymbol{O(E)}$ time.
   - b. A **Hamiltonian Cycle** of a directed graph, $G = (V,E)$, is a simple cycle that contains each vertex of G. Determining whether a directed graph has a Hamiltonian Cycle is **NPC**,

<inhon@mail.tku.edu.tw> May 31, 2015

P vs. NP

2. **Euler Tour vs. Hamiltonian Cycle**

- a. An **Euler Tour** traverses each edge of a directed graph $G = (V,E)$ exactly once, although it may visit a vertex more than once. If a graph has an Euler Tour we can find the edges in $O(E)$ time.
- b. A **Hamiltonian Cycle** of a directed graph, $G = (V,E)$, is a simple cycle that contains each vertex of G. Determining whether a directed graph has a Hamiltonian Cycle is **NPC**,

Approximation Algorithm

- 近似演算法（Approximation Algorithm）
  - 一個問題Q若經由上述証明方式，得知其屬於NP-complete問題，則代表此問題目前尚無有效率的演算法可以解決(即：無法在Polynomial Time內解決）。
  - 然而，某些屬於NP-complete的問題卻常常出現在各種領域!!若我們可退而求其次，去找尋一個**近似解**而非最佳解的話，則能夠預期以有效率的方式解決此問題。此即Approximation Algorithm的精神。

Approximation Algorithm

- 近似演算法（Approximation Algorithm）
  - 設計一個近似演算法需注意的Issue:
    - 近似演算法的時間複雜度要很低（至少要為Polynomial Time）
    - 需保証近似演算法所求出的解也是該問題的可行解
    - 在最差的情況下，用近似演算法所求出之近似可行解有多靠近最佳解

延伸閱讀

- http://en.wikipedia.org/wiki/Approximation_algorithm
- http://www.designofapproxalgs.com/
- http://www.win.tue.nl/~mdberg/Onderwijs/AdvAlg_Material/Course%20Notes/lecture5.pdf
- https://www.youtube.com/watch?v=hdch8ioLRqE
- https://www.youtube.com/watch?v=Dd0XNsAkkqE
- https://www.youtube.com/watch?v=f7U7UK6iiU4&list=PLTZbNwgO5eboxncIsmq95u_4nCtyziLKX

Tamkang University Software Engineering Group
淡 江 軟 體 工 程 實 驗 室
http://www.tkse.tku.edu.tw/

The End

累積在妳(你)身上的價值是
別人拿不走、破壞不了的

Reporter : Ying-Hong Wang
E-mail :inhon@mail.tku.edu.tw
<inhon@mail.tku.edu.tw>
May 31, 2015