COMP225: Algorithms and Data Structures

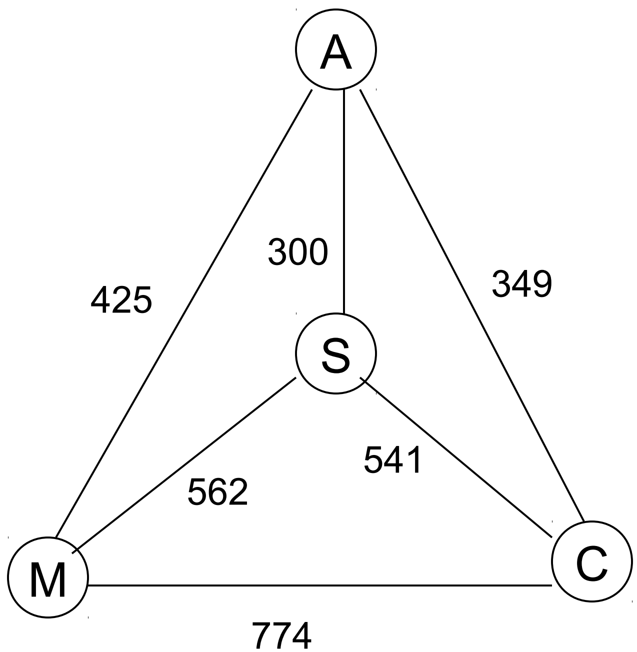
Applications of Graphs (2)

Mark Dras
Mark.Dras@mq.edu.au
E6A380

Outline

- Euler circuits
- Shortest paths
- Miscellanea

Aside: Travelling Salesman



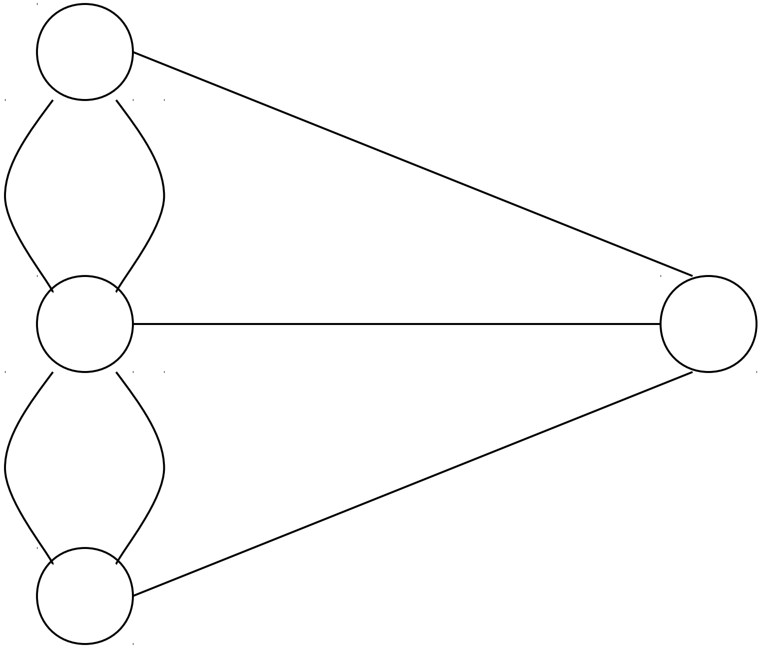
optimal: 1877 (A-C-S-M-A)

greedy: 2040 (A-S-C-M-A)

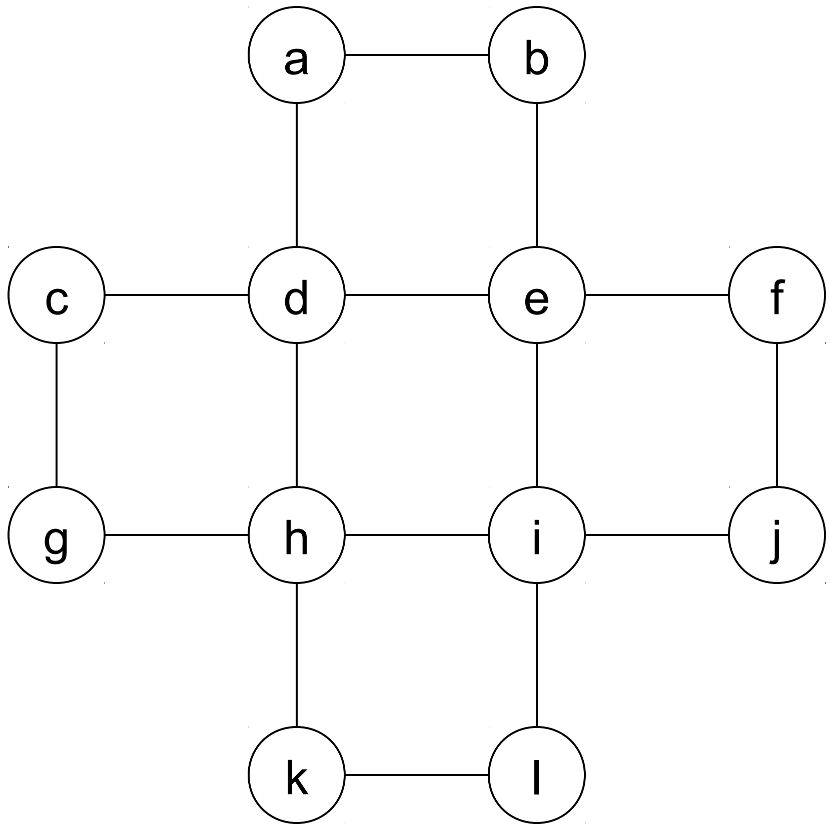
Euler Circuit

- Original problem: Seven Bridges of Königsberg
- Aim is to traverse every edge exactly once
  - doesn't matter how many times a particular vertex is visited
  - must end at start vertex
  - an Euler path is where you traverse each edge once but don't end up at the same vertex

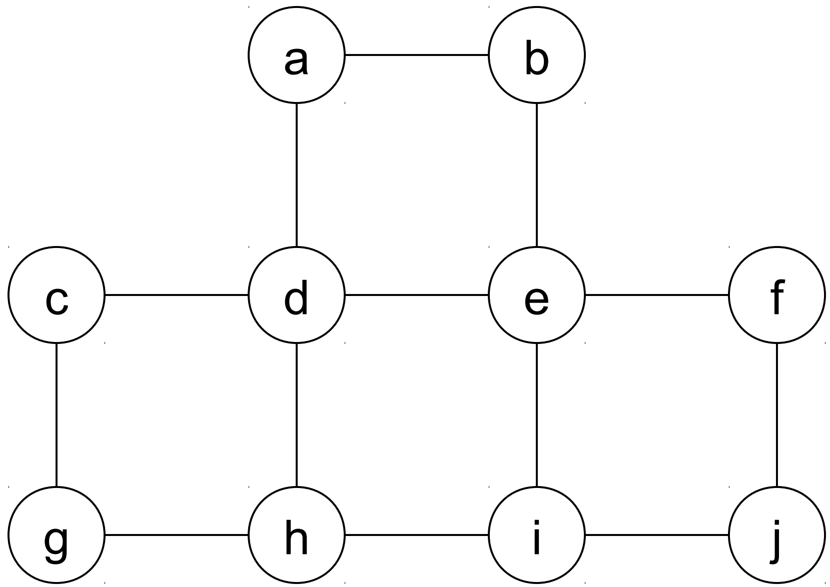
Euler Circuit



Euler Circuit



Euler Circuit



Euler Circuit

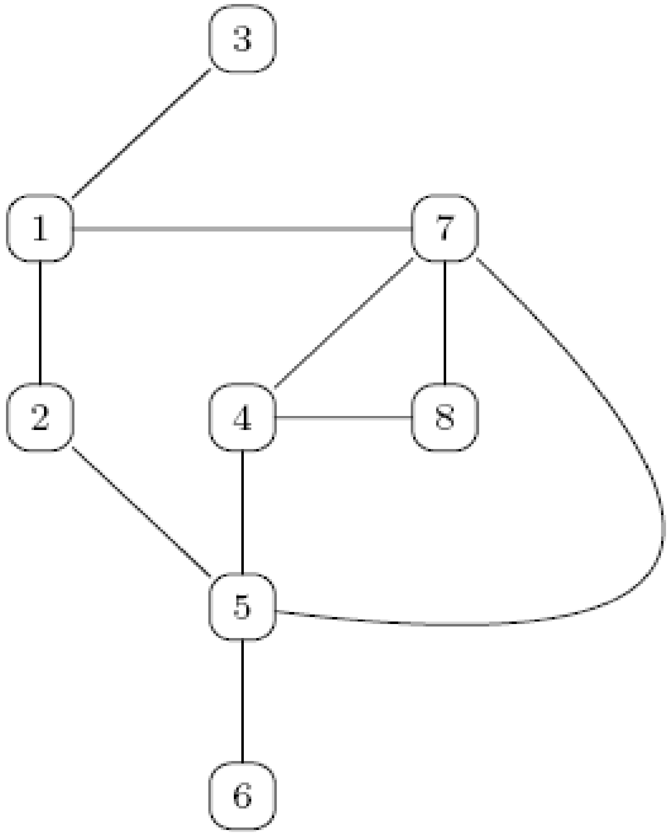
- Finding a cycle
  - an algorithm to find *whether* there is a cycle is easier
  - an Euler circuit is possible iff each node is of even degree
    - intuitively, this is because to traverse each edge you need to both enter and leave each node
  - an Euler path is possible iff
    - each node is of even degree, except for two of odd degree; or
    - an Euler circuit is possible

Euler Circuit

```
bool hasEulerCircuit(int v)
// detects whether there is an Euler circuit starting from vertex v

circuitSoFar = true
mark v as visited
if (degree(v) is odd)
    return false
for (each unvisited vertex u adjacent to v)
    circuitSoFar = circuitSoFar && isEulerCircuit(u)
return circuitSoFar
```

Euler Circuit: Exercise



Euler Circuit

- So obviously problem is not too difficult
- How about a problem where every vertex is visited once?
  - Hamiltonian circuit

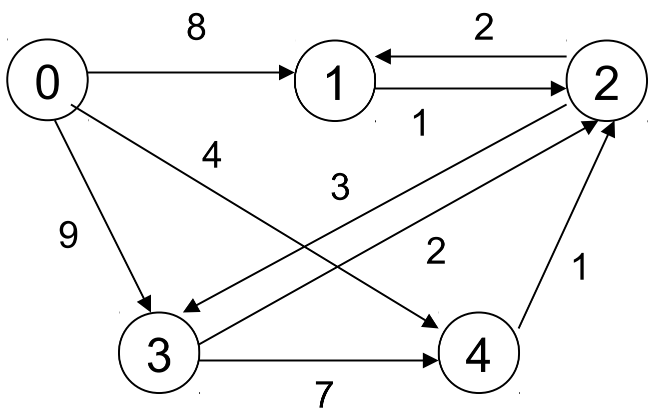
Outline

- Euler circuits
- Shortest paths
- Miscellanea

Shortest Path

- Aim to get from one vertex v_1 to another v_2 by the least cost path
- Note this isn't the same as the minimal spanning tree problem
  - that tries to connect all the nodes into a single least-cost tree
  - here we only look for a single connection between nodes, which must be a path
  - consider shortest path from *b* to *g* in Prim's Algorithm example

Shortest Path



	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

Shortest Path

- Shortest path from vertex 0 to vertex 1 has cost 7
  - not the edge (0,1): cost is 8
  - shortest path is 0-4-2-1
- Can we get this via a greedy algorithm?
  - in this case yes: take greedy choice at each step from 0
  - however, not for the general case

Shortest Path

- What's an alternative?
  - same as Travelling Salesman: enumerate every single possible path and evaluate weights
  - to find the shortest path from 0 to 1:
    - evaluate 0-1
    - evaluate 0-3-2-1
    - evaluate 0-3-4-2-1
    - evaluate 0-4-2-1
  - in general, computationally expensive

Dijkstra's Algorithm

- Determines shortest path between origin and all other vertices
- Algorithm uses:
  - a set vertexSet of selected vertices
  - an array weight, where weight[v] is the weight of the shortest path from vertex 0 to vertex v that passes through vertices in vertexSet
- Initially, vertexSet contains just 0, weight[v] is just adjMatrix[0][v]

Dijkstra's Algorithm

- Then
  - find a vertex *v* that is not in vertexSet that minimises weight[v]
  - add *v* to vertexSet
  - can you improve the cost to any other vertex *u* not in vertexSet by passing through *v*?
    - to do this, break path from 0 to *u* into two pieces:
    - weight[v] = weight of shortest path from 0 to v
    - matrix[v][u] = weight of edge from *v* to *u*

Dijkstra's Algorithm Example

			weight				
step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	-	0	0	8	∞	9	4
2	4	0,4	0	8	5	9	4
3	2	0,4,2	0	7	5	8	4
4	1	0,4,2,1	0	7	5	8	4
5	3	0,4,2,1,3	0	7	5	8	4

Dijkstra's Algorithm Example

- Step 1
  - vertexSet contains 0, weight is first row of adjMatrix (with weight[0] = 0)
- Step 2
  - weight[4] is smallest value in weight
  - add 4 to vertexSet
  - for each vertex not in vertexSet (i.e. 1, 2, 3), check if it's shorter to go via 4
  - shorter for 2 (weight[4] + adjMatrix[4][2] = 5)

Dijkstra's Algorithm Example

- Step 3
  - weight[2] = 5 is the smallest value in weight (ignoring weight[0] and weight[4])
  - add 2 to vertexSet
  - shorter to go via 2 for both 1 and 3
  - weight[1] = 7, weight[3] = 8

Dijkstra's Algorithm Example

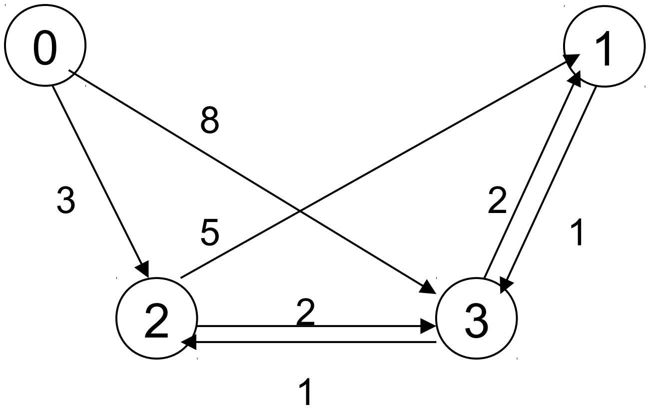
- Step 4
  - weight[1] = 7
  - add 1 to vertexSet
- Step 5
  - add only remaining vertex, 3, to vertexSet

Dijkstra's Algorithm

```
shortestPath(in theGraph: Graph, in weight:WeightArray)
// finds the min-cost paths between an origin vertex and all other vertices
// in a weighted directed graph; theGraph's weights are non-negative

create a set vertexSet that contains only vertex 0
n = number of vertices in theGraph
for (v = 0 through n-1)  weight[v] = matrix[0][v]
// invariant: weight[v] is the smallest weight to v from vertices in vertexSet
for (step = 2 through n) {
    find smallest weight[v] such that v is not in vertexSet
    add v to vertexSet
    for (all vertices u not in vertexSet)
        if (weight[u] > weight[v] + matrix[v][u])
            weight[u] = weight[v] + matrix[v][u]
}
```

Dijkstra's Algorithm: Exercise



Analysis

- In the form given, worst-case time complexity is $O(V^2)$
  - outer for loop loops through all |V| vertices
  - inner for loop loops through all vertices not yet in vertexSet (on average |V|/2)

A Better Variant

- Can improve by adding a global PQ, as with Prim's, storing vertices with distances
  - complexity is $O(E.T_{dk} + V.T_{em})$, where where T_{dk} and T_{em} are the complexities of the decrease-key and extract-minimum operations in the PQ, respectively
  - where a minheap is used, T_{dk} and T_{em} are both $O(\log n)$ for heap of size n
  - overall complexity then is $O(V \log V + E \log V)$

Dijkstra's Algorithm

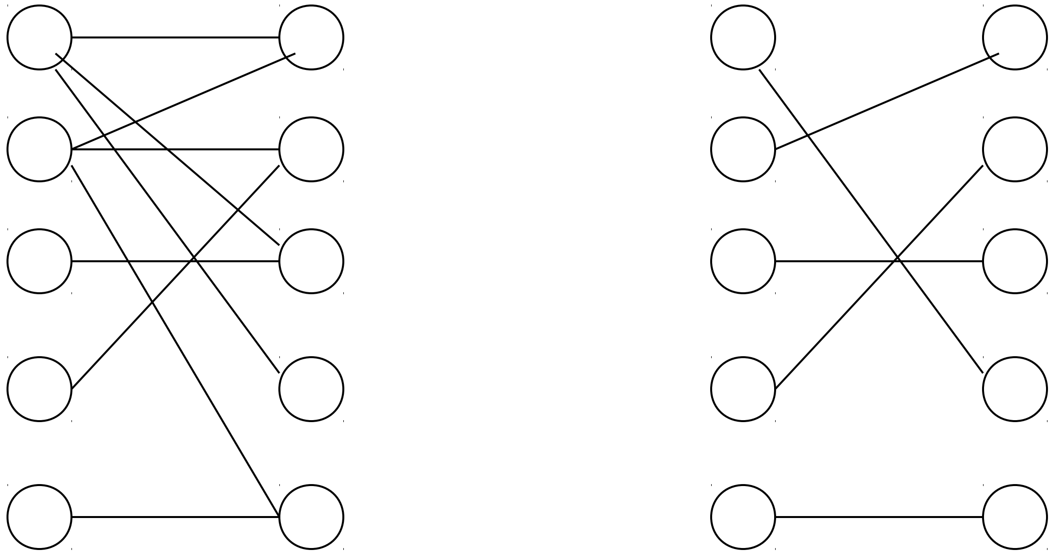
- Why don't we have to enumerate all solutions, as in Travelling Salesman?
  - the key is the loop invariant
  - we keep a record of the best solution so far: **optimal substructure**
  - this is a fundamental aspect of both greedy algorithms and dynamic programming
  - cf. divide-and-conquer

Outline

- Euler circuits
- Shortest paths
- Miscellanea

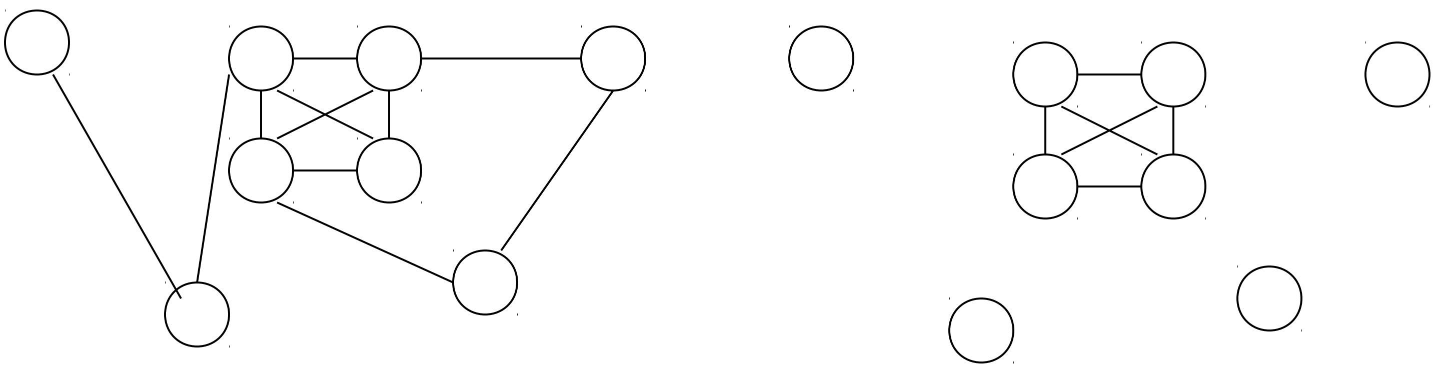
Other Graph Applications

- Matching
  - subgraph s.t. each vertex is incident to at most one edge



Other Graph Applications

- Clique detection



Real-Life Graph Problems

- Already mentioned ones
  - find best configuration for phone network links: minimal spanning tree
  - find best route for postman: weighted version of Euler circuit
  - scheduling of tasks: topological sort
  - assignment of jobs: matching

Real-Life Graph Problems

- In DNA sequencing: for a fragment *f*, some fragments are forced to go to the left of *f*, others to the right, others are free; need to find a consistent ordering
  - Create a directed graph with links between constrained fragments, then do topological sort

Real-Life Graph Problems

- In porting code from Unix to DOS, filenames can only be 8 characters, but must be unique. Can't just truncate (e.g. *filename1* vs *filename2*)
  - Each original filename becomes a vertex, as does the set of acceptable renamings. Want n edges with no vertices in common: matching.

Real-Life Graph Problems

- In organised tax fraud, fake tax returns are similar but not identical. How to detect clusters of similar forms?
  - Each form is a vertex, with an edge between them if they match some percentage of entries: then identify clique

Real-Life Graph Problems

- In OCR, need a way to separate lines of text. Although there's some white space between lines, problems like noise and page tilt make it hard to find. How to do line segmentation?
  - Each pixel is a vertex, edges between neighbouring pixels. Weight is proportional to darkness: find shortest path.