



Murdoch
UNIVERSITY

Data Structures and Abstractions

Hash Tables & Graphs

Lecture 11



Important Advice for LAB/Assignment

- You must complete Lab 10.
 - Submission needed for the last assessed lab
 - Submission needed for the assignment
- Your BST needs to be usable beyond the purposes of the lab/assignment.
- Follow the assignment specifications carefully. Read the QandA file (when available) regularly to see any clarification or advice.
 - If the answer to your question is not there, ask early.
- Be mindful of summing small floating point numbers. Errors accumulate.
 - See the following for some advice:
 - https://en.wikipedia.org/wiki/Kahan_summation_algorithm
 - For a more detailed answer see: “What Every Computer Scientist Should Know About Floating-Point Arithmetic” at http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Maps

- Previously we have looked at the STL map class, where any type can be used as a key into a container of paired values, giving direct access to the second part of the data.
- So we can have:

```
map<string, string> DictionaryType;
```

```
Dictionary dictionary; // not really a good way to name
```

```
dictionary["aardvark"] = "A nocturnal mammal of southern  
Africa"
```

```
cout << "arardvark:" << dictionary["aardvark"];
```

```
//Figure out how may string object constructions occurred in the lines  
//above
```

Hash Tables

- One way to achieve this kind of direct access for the map class is to use what is known as a *hash table*.
 - If keys are unique a balanced binary search tree can be used.
 - If keys are not unique and key are unordered as in `std::unordered_multimap` or `std::unordered_multiset` , then hashing is used.
- When storing the data, the key—in this case “aardvark” —is passed through a *hash function*, to give an index into an ordinary array. [1] [2]
- The quality of the hash function determines how many different keys hash to the same index value. (technically known as “collisions”)
- No hash function is perfect under all conditions, therefore there will always be clashes (“collisions”).
- Therefore there must also be a *collision resolution* defined.
- Hash tables will always have empty space. To work most efficiently they are generally required to be no more than half full.

Dealing with Strings

- The key used in the above example is a string.
- Obviously you cannot pass a string through a mathematical function.
- Therefore strings must be mapped to integers before hashing.
- There are many ways to do this, however it is important to make sure that the method chosen does not promote collisions.

Insertion into a Hash Table

- Insert (pair)
- $\text{index} = \text{HashFunction}(\text{pair.key})$
 $\text{index} = \text{index} \bmod \text{arraySize}$ // in hash table i.e %

IF $\text{array}[\text{index}]$ is empty

$\text{array}[\text{index}] = \text{pair}$

ELSE

HandleCollision (index, pair)

ENDIF

- End Insert

Searching a Hash Table

```
Search (key, found)
```

```
    index = HashFunction (key)
```

```
    index = index mod arraySize
```

```
    IF array[index] is empty
```

```
        found = false
```

```
    ELSE
```

```
        IF array[index].key = key
```

```
            found = true
```

```
        ELSE // another key was hashed to the same index
```

```
            found = CollisionSearch (index, pair)
```

```
        ENDIF
```

```
    ENDIF
```

```
End Search
```

Hash Functions

- The ideal function would: [1]
 - be easy to calculate;
 - never produce the same index from two different keys;
 - spread the records evenly throughout the array;
 - deal with ‘bad’ keys better than others.
- Of course no function has all of these attributes under all conditions.
 - It may be possible under restricted conditions where all keys are known in advance.
- Common Hash Functions
 - Truncation
 - Extraction
 - Folding
 - Modular arithmetic
 - Prime number division
 - Mid-square hashing
 - Radix conversion

Radix Conversion

- The best known of these is Radix Conversion.
- Choose a low prime [1] number such as 7, 11 or 13 to use as the base of a polynomial. Then use the digits of the key as the factors of the polynomial.
- Finally modulate by the array size, which should be a prime number.

- For example, if

key = 32934648

array size = 997

Base = 7

Then

$$\begin{aligned} \text{index} &= (3 * 7^7 + 2 * 7^6 + 9 * 7^5 + 3 * 7^4 + 4 * 7^3 + 6 * 7^2 + 4 * 7 + 8) \text{ MOD } 997 \\ &= 2866095 \text{ MOD } 997 \\ &= 717 \end{aligned}$$

Collision Resolution

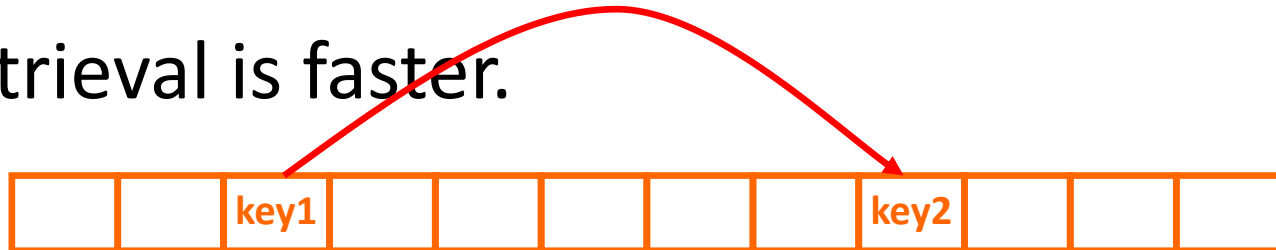
- Collision resolution needs to:
 - avoid clustering of records;
 - be as simple to code as possible;
 - only fail when the array is actually full;
 - be ‘reversible’ to allow for deletion/search.
- As before, no method fulfils all these requirements under every possible condition.
- Common collision resolutions are:
 - Linear probing
 - Quadratic probing
 - Random probing
 - Linked collisions
 - Overflow containers

Probing

- **Linear** probing simply looks for the **next empty space** in the array. So if index is full, index+1 is checked, then index+2, index+3 etc. This has the disadvantage of increasing clustering and therefore collisions.
- **Quadratic** probing looks for the next empty space using 'square' jumps. So if index is full, index+1 is checked, then index+4, index+9, index+16 etc. This avoids the clustering of linear probing, but can fail when the array is not full.
- **Random** probing uses a random number generator—from a set starting point—for the increments in index. This avoids clustering, but is harder to reverse when a record is to be deleted (search?). Use pseudo-random number generator.

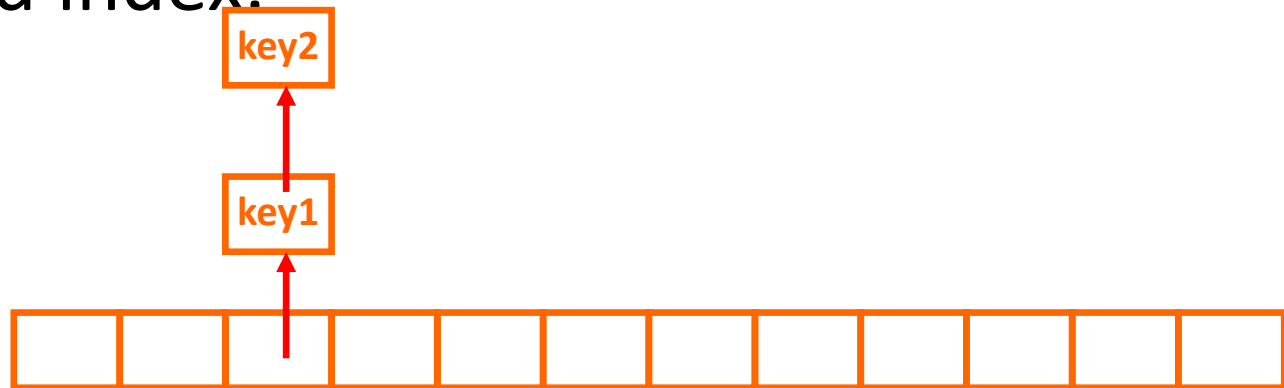
Linking Collisions

- After the first collision, a second hash function is used to generate an alternate position and the two are linked.
- A third collision would then have a link from the second collision and so on.
- This is harder to code and uses extra memory, but retrieval is faster.



An Overflow Container

- Instead of using a one dimensional array to store data, a two dimensional structure is used.
- The records are placed in a linked list from the hashed index.



Readings

- Reference book, Introduction to Algorithms. Chapter on Hash Tables.

Further Exploration

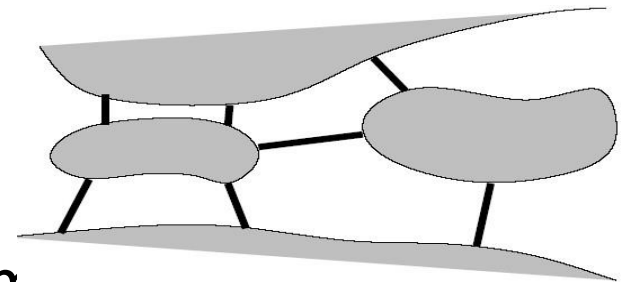
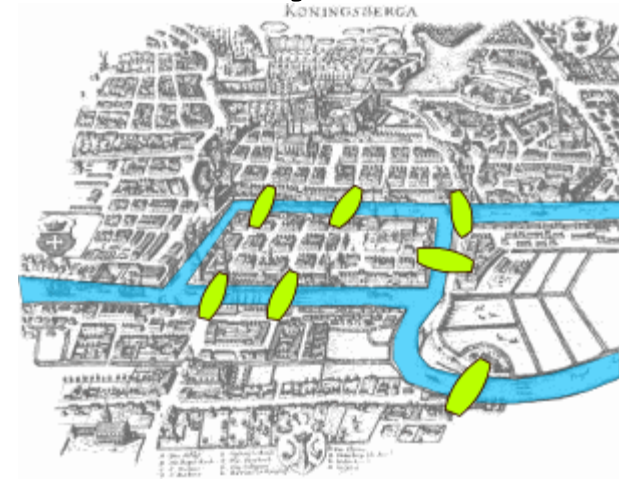
- Khan Academy Video one particular example of the use Hash functions [“Bitcoin: Cryptographic hash functions”](#)
- Tutorial on Hash functions
<http://research.cs.vt.edu/AVresearch/hashing/>

Graph Theory



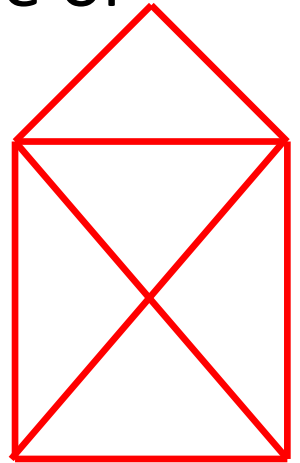
The Origins of Graph Theory

- Graph Theory (unlike a lot of what we do) dates back to before 1736.
- In Konisberg there were two islands in the middle of a river, connected by 7 bridges. [1]
 - Textbook has the abstract version.
- The question was: “is it possible to cross each bridge exactly once?”
 - Abstract representation is used to investigate solutions.
 - Any solution obtained can then be used for similar problems.
- In 1736, Euler answered this problem, by establishing “Graph Theory” as a discipline. (The answer is “no”)



Another Common Problem

- As a child you may have met something similar: Draw the shape below without taking your pen off the page and without going over any line or node more than once.



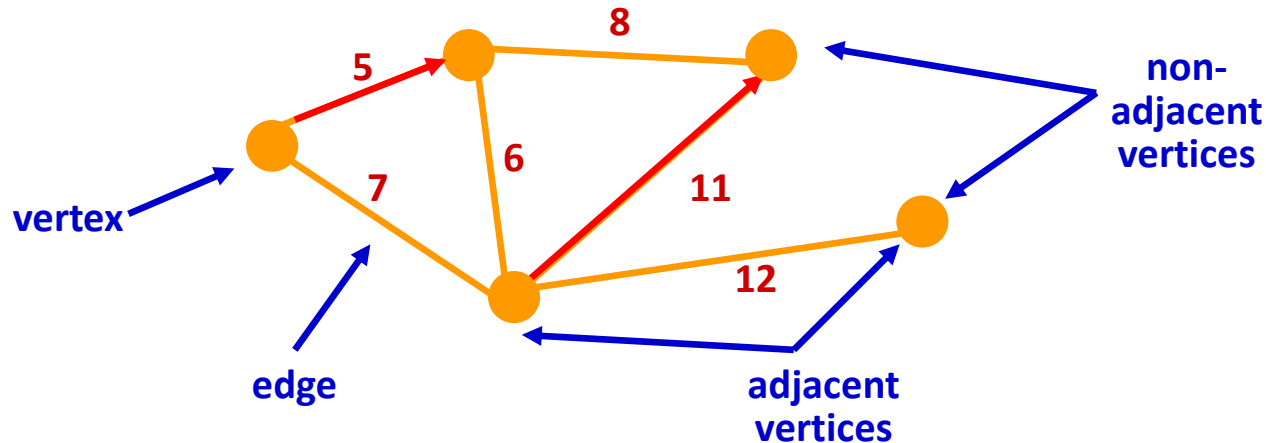
- It is a graph problem, just as the Konisberg problem is a graph problem.

But...

- But the theory itself remained a kind of mathematician-only esoteric field until
 1. Computers became available that could handle graph processing algorithms in reasonable time.
 2. Many of the complex problems of society were recognised to be graph problems.
 3. It was realised that Network traffic and the WWW were graphs.
 4. Some AI applications (simulations, neural networks etc) were discovered to use graph theory.
 5. Computer game playing required graph theory.

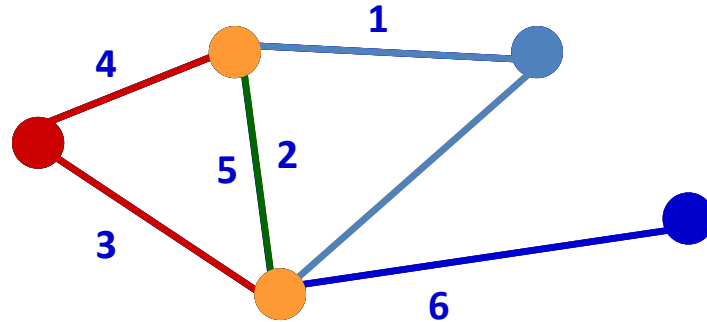
So What is a Graph?

- A graph is a set of *vertices* connected by *edges*.



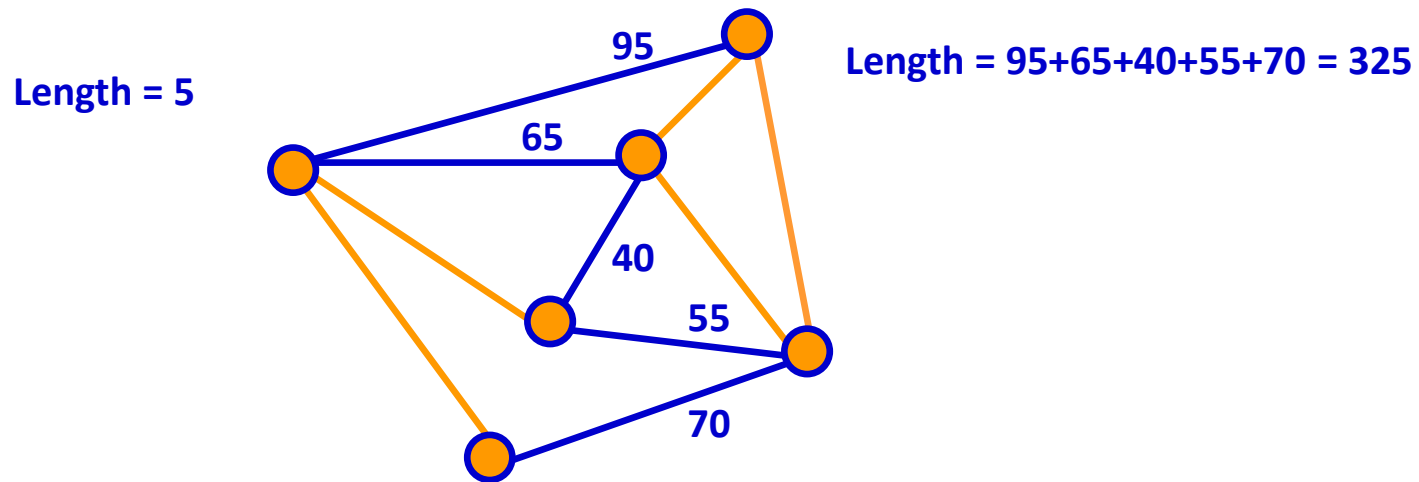
- Two vertices are *adjacent* if they are connected by a single edge.
- A graph is *weighted* if there is a number associated with each edge. (can be cost, distance, ..etc)
- A graph is *directed* if any of the edges are one-way.

Graph Definitions

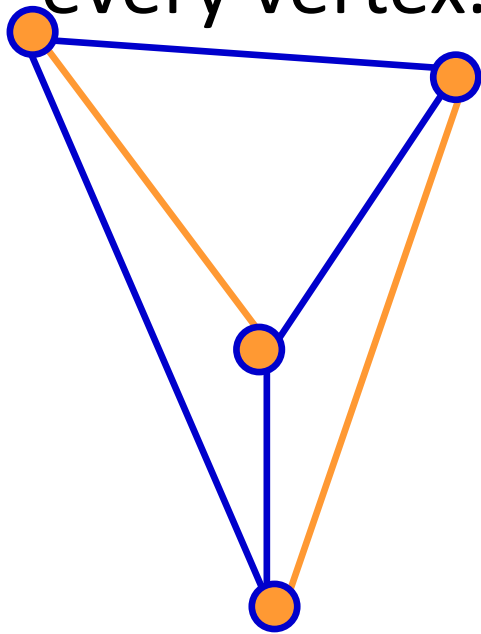


- A *path* is a sequence of adjacent vertices
- A *simple path* is one that has distinct edges: no vertex is visited twice.
- A *cyclic path* is one where the start and finish are the same vertex.
- Two paths are *disjoint* if they have no vertices in common, other than, possibly, their endpoints. [see the red and blue paths]

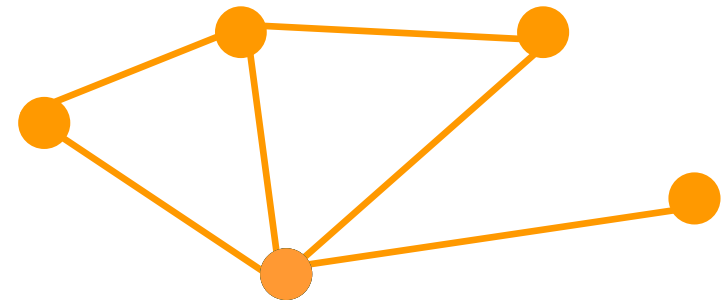
- In an unweighted graph, the length of a path is the number of traversed edges.
- In a weighted graph, the length of a path is the sum of the weights of the traversed edges.



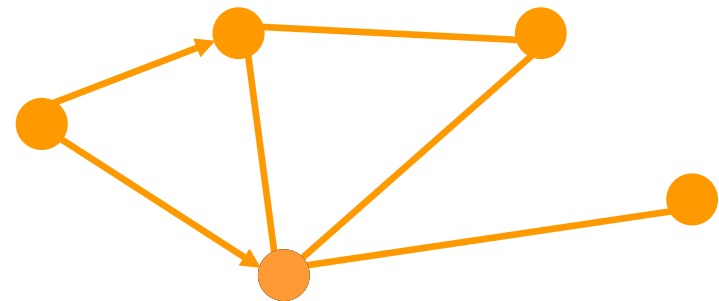
- A *tour* is a cyclic path that touches every vertex.



- A connected graph is one where every vertex is reachable from every other vertex

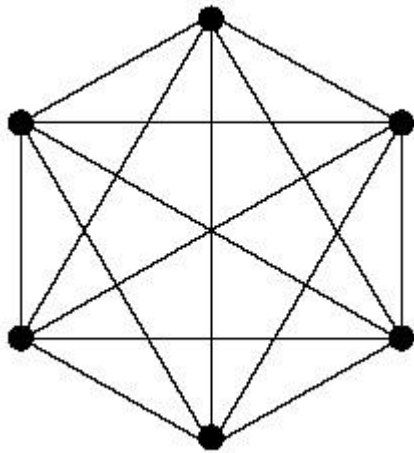


Connected

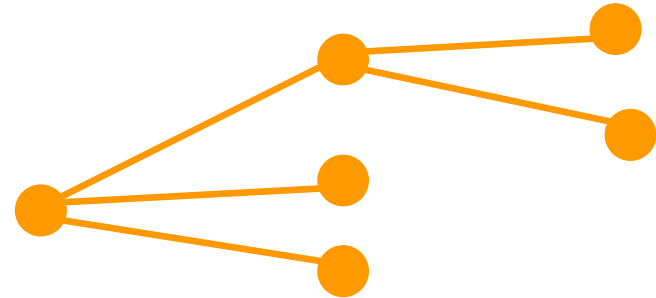


Not connected

- A *complete* graph is one where every vertex is adjacent to every other vertex.



- A graph with no cycles (an *acyclic* graph) is a tree.

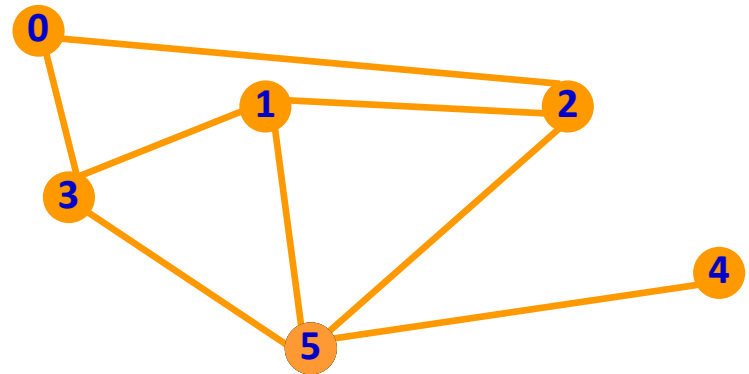


Data Structures to Represent Graphs

- Representing a graph as vertices and edges is fine in the abstract (physical) sense but makes processing too difficult.
- Two alternatives are therefore used within programming:
 - Adjacency matrices
 - Constant access time
 - Slow search time
 - Adjacency list
 - Fast search time
 - Slow access time
- For both of these, the vertices are arbitrarily numbered.

Adjacency Matrix Representation

- The graph is represented as a two dimensional array of boolean.
- A vertex is not considered to connect to itself.

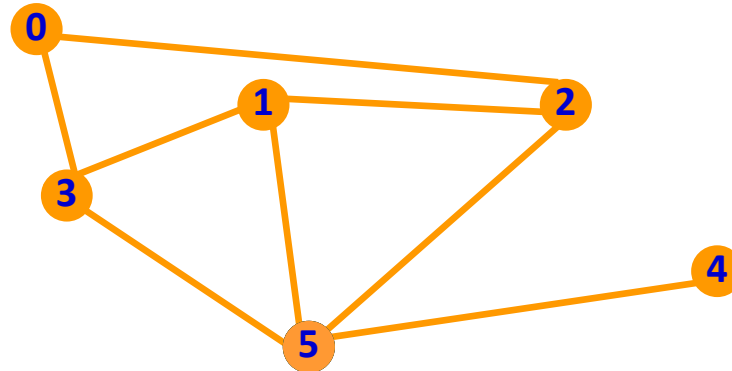


	0	1	2	3	4	5
0	false	false	true	true	false	false
1	false	false	true	true	false	true
2	true	true	false	false	false	true
3	true	true	false	false	false	true
4	false	false	false	false	false	true
5	false	true	true	true	true	false

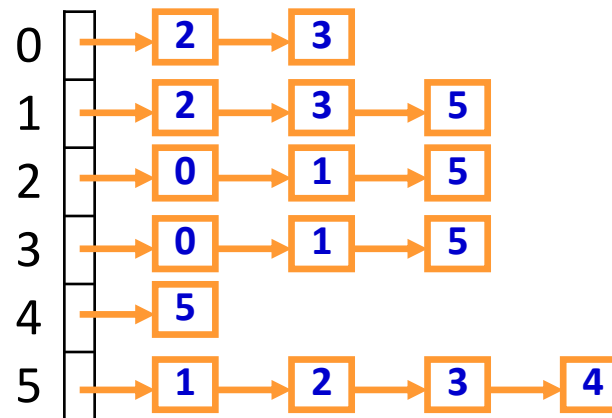
Drawing an Adjacency Matrix

- Make sure you can draw an adjacency matrix for a graph. Use the Graph program to check your answers.

Adjacency List Representation



- The graph is represented as a one dimensional sorted list of connected vertices:



Drawing an Adjacency List

- Make sure you can draw an adjacency list for a graph. Use the Graph program to check your answers.

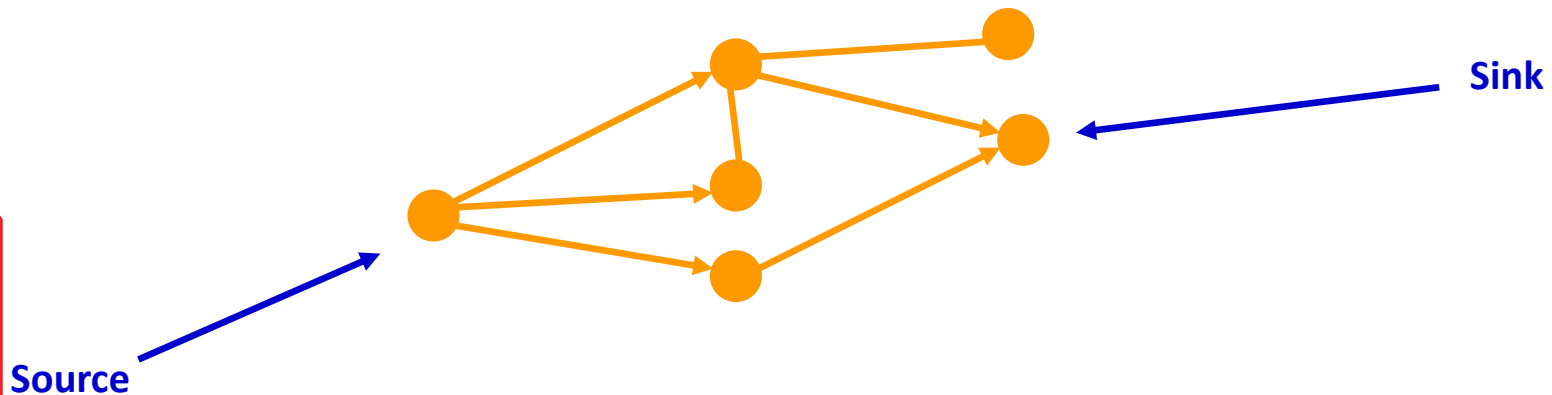
Matrix and List Comparison

- Advantages of Lists
 - More flexible as the size is not fixed
 - Less space used: $O(V+E)$ rather than $O(V^2)$ for a matrix.
 - Faster processing (searching) at each vertex
- Advantages of Matrices
 - Easier to program
 - Access time to find out if a pair of vertices are connected is constant time as opposed to $O(V)$ for lists.

Directed Graph Definitions

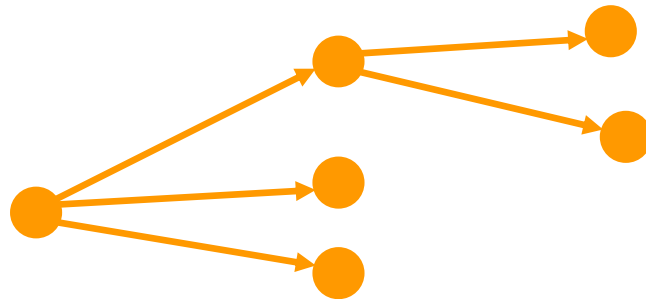
- Directed graphs are also known as di-graphs.
- A vertex is *reachable* from another vertex if there is a path between them.
- It is assumed that each vertex can reach itself.
- The *in-degree* of a vertex is the number of edges leading into a vertex.
- The *out-degree* of a vertex is the number of edges leading out of a vertex.

- A *sink* is a vertex with out-degree of zero.



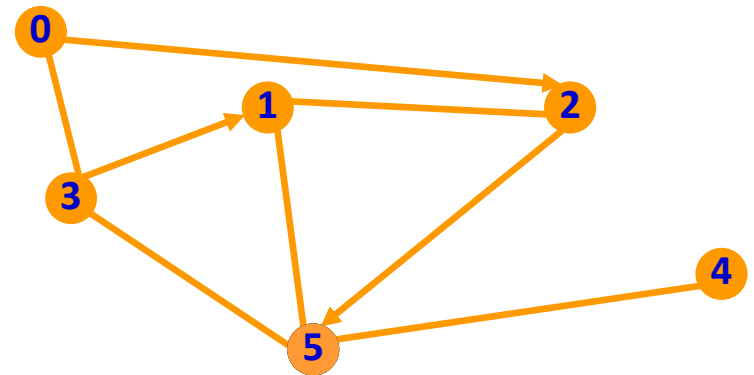
- A *source* is a vertex of in-degree 1: it is reachable only from itself.

- A *map* is a di-graph where every vertex has out-degree 1.
- A di-graph is *strongly connected* if every vertex is reachable from every other vertex.
- A di-graph with no cycles is an Acyclic Directed Graph, or DAG.



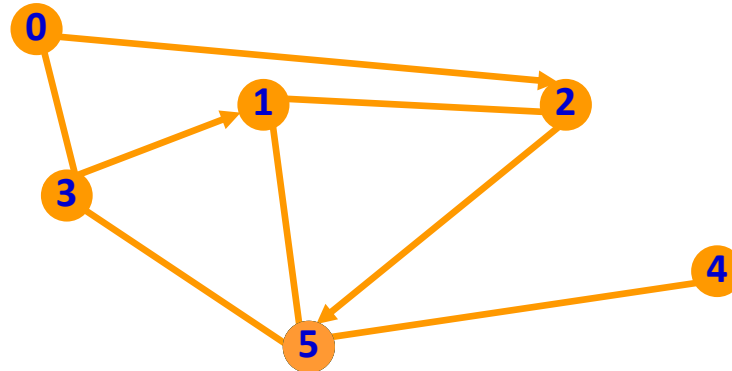
Adjacency Matrix Representation of a Di-graph

- The di-graph is represented as a two dimensional array of boolean.
- Note that in a di-graph vertices are considered to be connected to themselves.

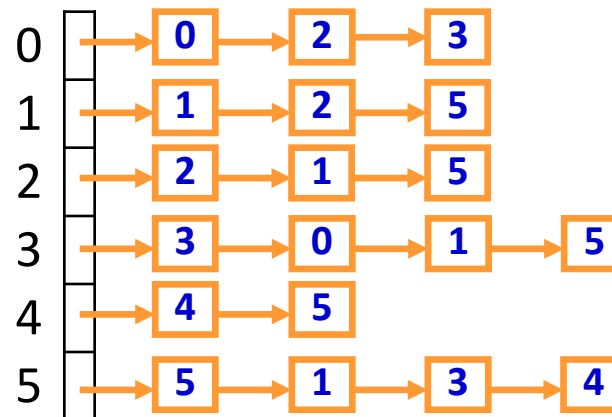


	0	1	2	3	4	5
0	true	false	true	true	false	false
1	false	true	true	false	false	true
2	false	true	true	false	false	true
3	true	true	false	true	false	true
4	false	false	false	false	true	true
5	false	true	false	true	true	true

Adjacency List Representation



- The di-graph is represented as a one dimensional sorted list of connected vertices:



Graph Domains

- Social media – friendship networks
 - Interconnections in ecosystems
 - Genetics and ancestry
 - Chemical structures
 - Traversal problems
 - Travel itineraries
 - Neural networks
 - The WWW (the biggest graph of them all)
 - Electric circuits
 - Scheduling
 - Financial transactions
 - Compilers use graphs to represent call structures
 - Within games software
 - UML diagrams, data flow diagrams, E-R diagrams etc
 - Automatic diagram generation
- etc.

Some Graph, Di-Graph and DAG Processing Problems

- Searching: how do we get from a particular vertex to another.
- Connectivity: is a given graph connected.
- Find the minimum length set of edges that connects all vertices (the Minimum Spanning Tree or MST).
- Find the shortest path between two vertices.
- Find the shortest path from a specific vertex to all other vertices (the Shortest Path Tree or SPT).
- Planarity: can a specific graph be drawn without any intersecting lines?
- Matching: what is the largest subset of edges with the property that no two are connected to the same vertex?
- Find the tour with the shortest path (mail carrier problem).
- Topological Sort: sort the vertices of a DAG in order of the number of dependencies.

Complex problems

- Graphs are a powerful tool for modelling complex problems.
- *“The great unexplored frontier is complexity – I am convinced that nations and people that master the new science of complexity will become the economic, cultural, and political superpowers of the next century.”* Heinz Pagels

In Fact

- These problems are NP-Hard.
- There is no solution for any of them that is guaranteed to be solvable in a reasonable amount of time.
 - Restricted case solutions are possible but not in the general case.
- There are only solutions that work quite well in some circumstances.
- This, combined with the large number of domains, makes this field one that is rich in research possibilities.

Readings

- Textbook Chapter on Graphs.
- The lecture notes and textbook is sufficient for this unit.
- Further exploration:
 - [Complex systems: Network thinking](#), Melanie Mitchell, Artificial Intelligence, vol. 170(18), Science Direct, Elsevier.
 - Reference book, Introduction to Algorithms. Part on Graph Algorithms contains a number of chapters on graph algorithms. (for further study)

Graph Processing Algorithms



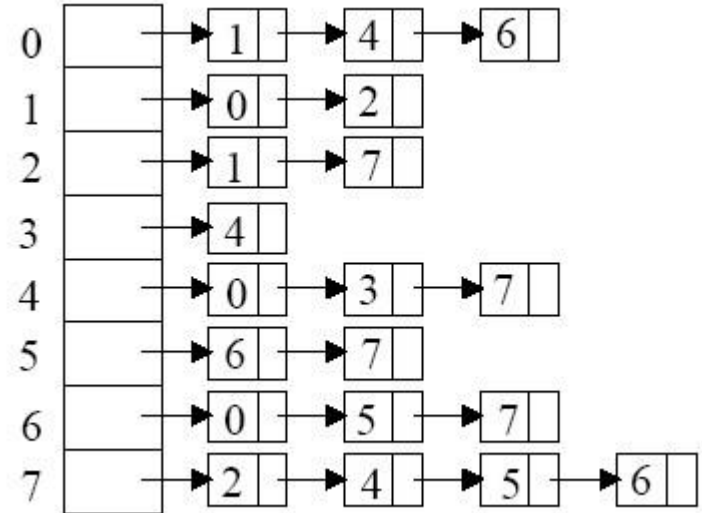
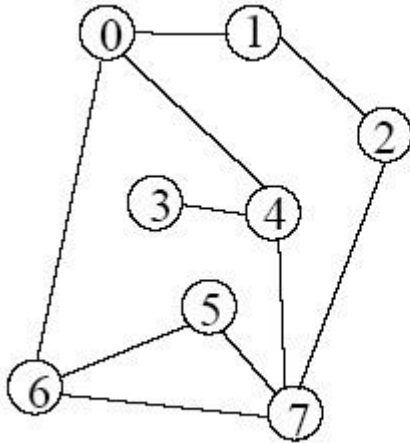
Depth First Search

- Depth first search (DFS) answers the question “is vertex A connected to vertex B?”
- Using an adjacency matrix, DFS takes $O(V^2)$.
- Using an adjacency list, DFS takes $O(V+E)$.
- DFS of a graph is usually done recursively.
- Textbook has code with explanations.

Depth First Search Algorithm

- DFS (fromVertex, toVertex)
- boolean found = false
- IF fromVertex \neq toVertex // what if == ?
- IF the fromVertex has not already been visited
- Mark it as visited
- FOR each vertex in its adjacency list
- AND while not found
- found = DFS(adjacentVertex, toVertex)

Examples



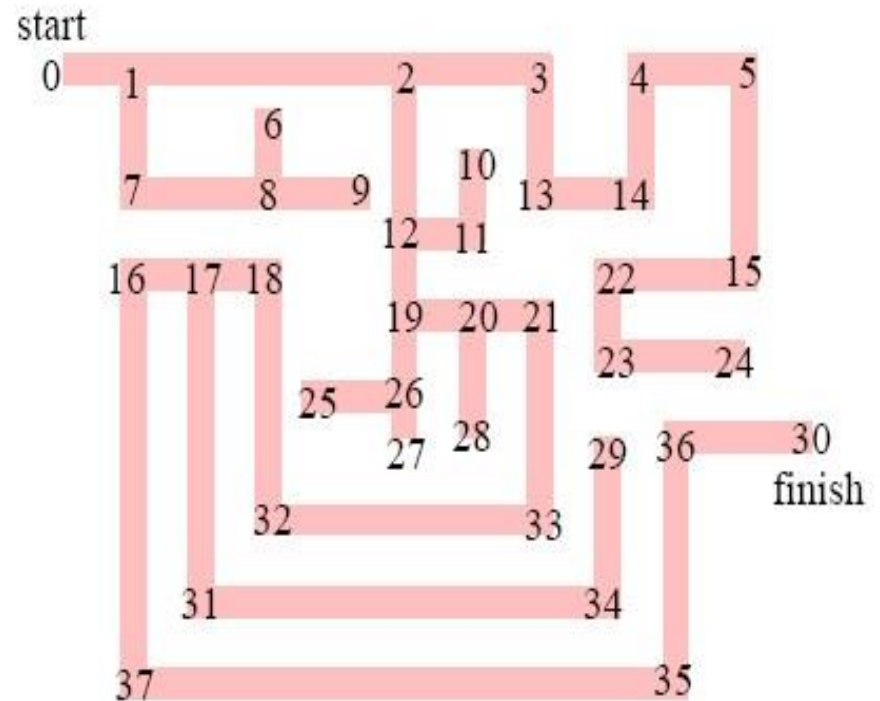
Starting From	Searching For	Vertices Visited	Actual Path
0	5	0, 1, 2, 7, 4, 3, 5	0, 1, 2, 7, 5
3	1	3, 4, 0, 1	3, 4, 0, 1
6	2	6, 0, 1, 2	6, 0, 1, 2
2	6	2, 1, 0, 4, 3, 7, 5, 6	2, 1, 0, 4, 7, 5, 6

Other Examples

- Make sure you can do, by hand, a depth first search of a graph. The Graph program can be used to check your answers.

Application to Maze Solving

- Consider a maze.
- This can be modelled as a graph.
- DFS can now be used to find the route from start to finish.

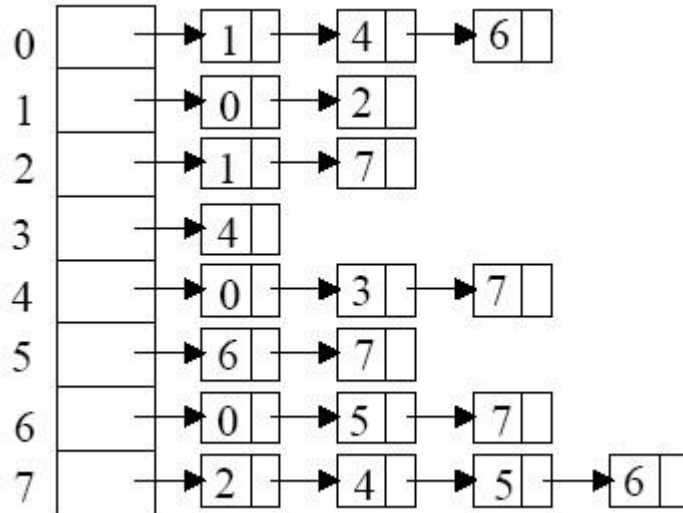
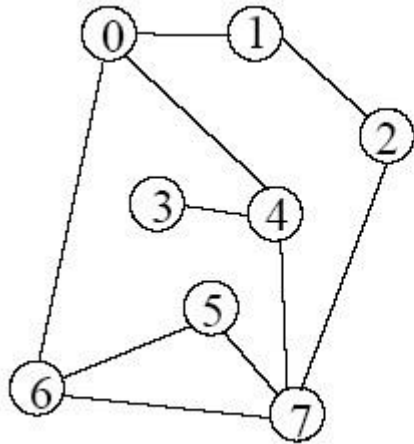


Breadth First Search

- Breadth First Search (BFS) finds the closest solution to your current position in the graph.
- For example, in all game playing you want the fastest route to a game win.
- In a tree it is equivalent to searching layer by layer. [Go through the animation of the breadth first search in the tree lecture notes first.](#)

- Breadth First Search (fromVertex, toVertex)
- boolean found = false
- IF (fromVertex <> toVertex)
- Put fromVertex on a **queue**
- Mark fromVertex as visited
- WHILE the queue is not empty AND not found
- Remove qVertex from the queue
- FOR each aVertex in the adjacency list of qVertex AND not found
- IF aVertex has not been visited
- IF aVertex <> toVertex
- Add aVertex to the queue
- Mark aVertex as visited
- ELSE
- found = true
- ENDIF
- ENDIF
- ENDFOR
- ENDWHILE
- ELSE
- found = true
- ENDIF
- return found
- END BFS

Breadth First Search Examples



Starting From	Searching For	Vertices Visited	Actual Path
0	5	0; 1, 4, 6; 2, 3, 7, 5	0, 6, 5
3	1	3; 4; 0, 7; 1	3, 4, 0, 1
6	2	6; 0, 5, 7; 1, 4, 2	6, 7, 2
2	6	2; 1, 7; 0, 4, 5, 6	2, 7, 6

Other Examples

- Make sure you can do, by hand, a breadth first search of a graph. Use the graphs program to check your answers.

Shortest Path Problems

- Shortest path problems apply to weighted graphs only.
- There are three sub-problems:
 - Find the shortest path from vertex X to vertex W .
 - Find the shortest path between every pair of vertices.
 - Find the shortest path from a particular vertex to all other vertices.
- The latter of these forms a Shortest Path Tree (SPT).

SPT Algorithms

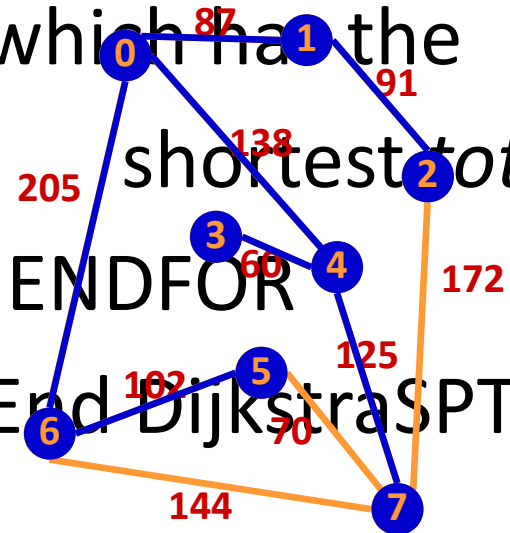
- There are many SPT algorithms.
- Some are faster than others and some are easier to code (never the same algorithm of course!)
- Research continues in the area because of the huge importance of network traffic routing, and other similar problems.
- We will look at only one as it is easy to understand and good enough to be useful.
- It has complexity of $O(V)$, which is very good for a graph theory algorithm.

Dijkstra's SPT Algorithm

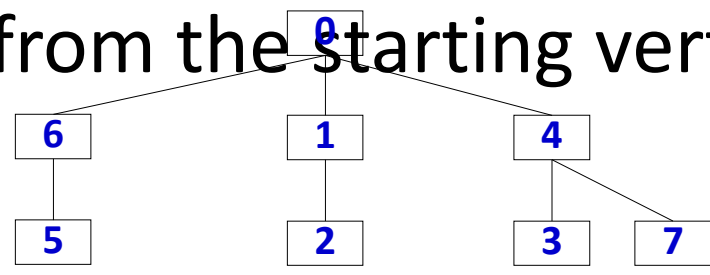
- DijkstraSPT
- Put the starting vertex into the tree
- FOR V-1 times
- Add the vertex that is adjacent to the SPT and

which has the

shortest *total path* from the starting vertex



From Vertex 0



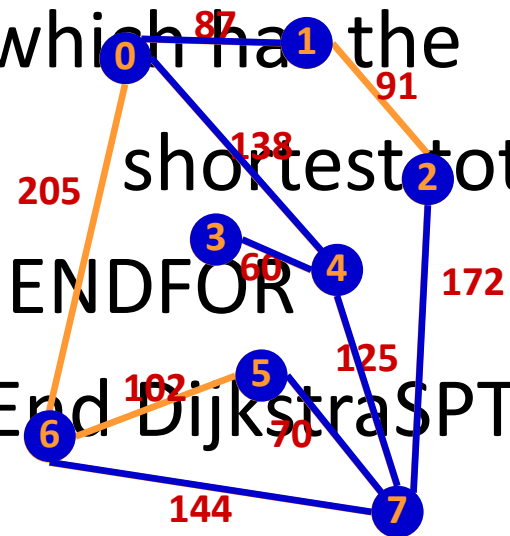
ENDFOR
End DijkstraSPT

Dijkstra's SPT Algorithm

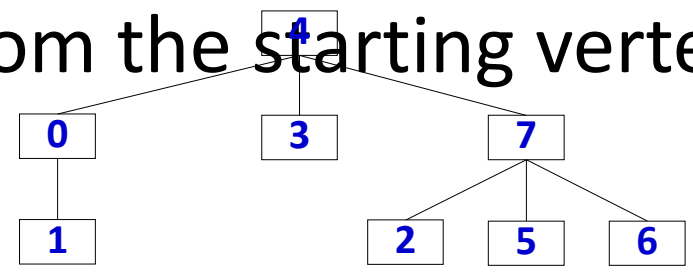
- DijkstraSPT
- Put the starting vertex into the tree
- FOR V-1 times
- Add the vertex that is adjacent to the SPT and

which has the

shortest total path from the starting vertex



From Vertex 4



Note that this tree is different from the previous one

Shortest Path Between Two Vertices

- This can be done using Dijkstra's SPT algorithm, but stopping when the target vertex is reached.

ShortestPath

Put the starting vertex into the tree

WHILE we have added less than $V-1$ vertices AND
target not found

 Add the vertex that is adjacent to the SPT and
 which has the shortest total path from the
 starting vertex

 IF this is the target vertex
 found = true

 ENDIF

ENDWHILE

Return found

END Shortest Path

Other Examples

- Make sure you can draw, by hand, an SPT of a graph from any starting vertex. Use the Graph program to check your answers.

Readings

- Textbook Chapter on Graphs.
- Reference book, Introduction to Algorithms. Part on Graph Algorithms contains a number of chapters on graph algorithms. Please read for further study. The lecture notes and textbook is sufficient for this unit.

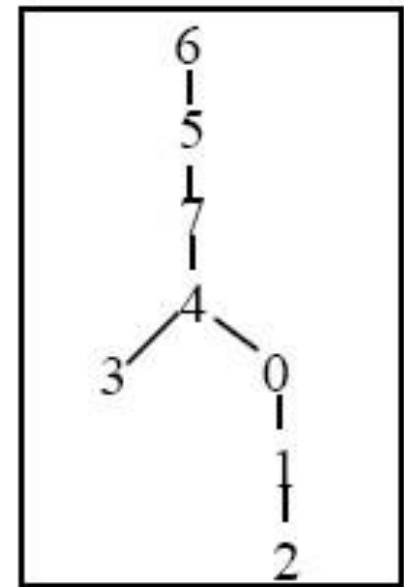
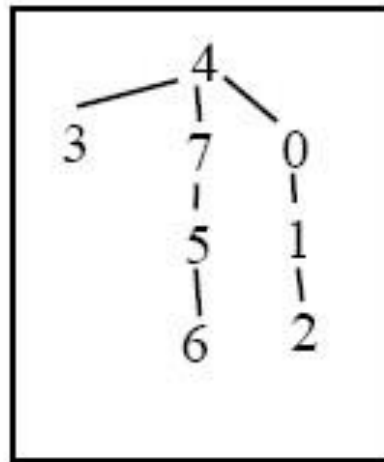
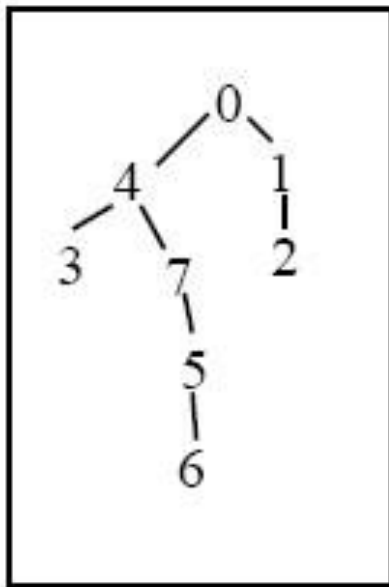
Minimum Spanning Tree Algorithms

MST versus SPT

- The SPT problem of the previous lecture involved finding the shortest path from a single vertex to every other vertex.
- The MST problem involves finding the shortest way to connect all the vertices to each other, using any vertex as a starting point.
- Both apply to weighted graphs only.
- The SPT is different for each starting vertex, the MST is the same no matter which vertex you start at.

Minimum Spanning Trees

- The starting point of an MST is irrelevant: all three of the trees below are equivalent and represent the MST of the same graph.

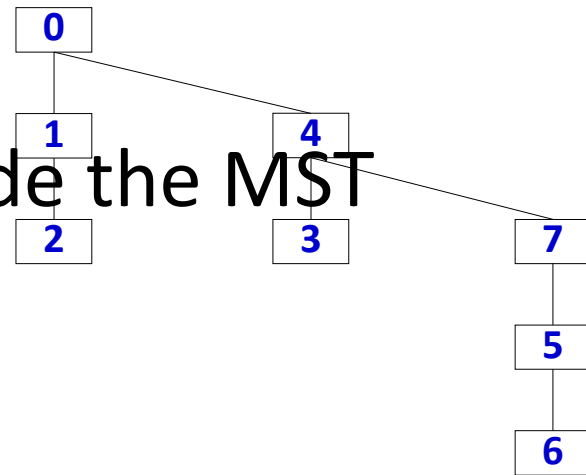
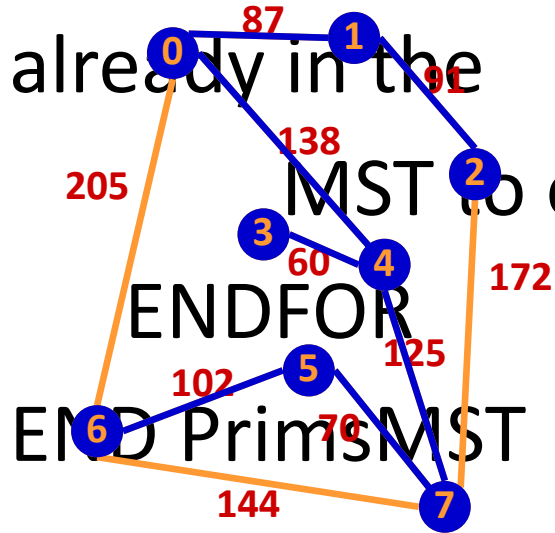


MST Algorithms

- There is much continuing research in this area.
- All algorithms have advantages and disadvantages.
- All algorithms are more or less efficient depending on the type of graph being processed.
- The choice of data structure and programming language also affects the speed.
- We will look at two algorithms: Prim's, Kruskal's and point out another one by

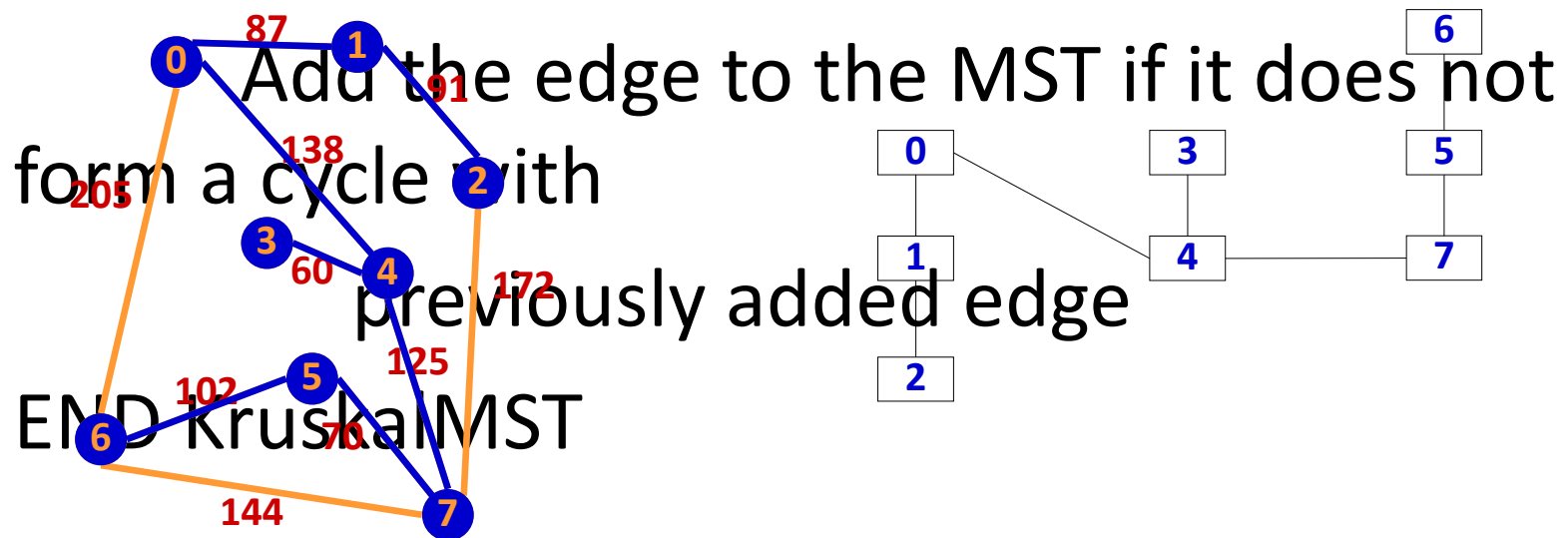
Prim's Algorithm (animation)

- PrimsMST
- Pick any vertex and put it in the MST
- FOR V-1 times
- Add the shortest edge from a vertex



Kruskal's Algorithm (animation)

- KruskalMST
- Sort the edges from shortest to longest
- LOOP for each edge from shortest to longest
- AND edges added $< V-1$



Note that this tree is topgraphically the same as the previous one.

Costs

Algorithm	Cost	Comment
Prim's	$O(V^2)$	Optimal for dense graphs
Kruskal's	$O(E \log(E))$	The highest cost is in the sorting
Boruvka's	$O(E \log(V))$	This is a conservative upper bound

- The algorithms will all run faster in some circumstances.
- There is no algorithm that is always fastest.
- The faster the algorithm, generally the harder it is to code, and therefore the more prone to error.
- There is no algorithm that can guarantee

Other Examples

- Make sure you can draw, by hand, an MST using each of these algorithms. The Graph program will allow you to check your answers.

Readings

- Textbook Chapter on Graphs.
- Reference book, Introduction to Algorithms. Part on Graph Algorithms contains a number of chapters on graph algorithms. Please read for further study. The lecture notes and textbook is sufficient for this unit. The reference book would give more details.