PROG 20799
DATA STRUCTURES AND ALGORITHMS - C

Simon Hood

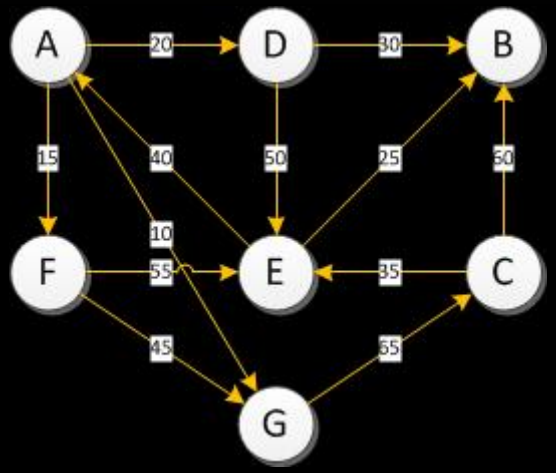
simon.hood@sheridancollege.ca

OUTLINE

- Traversing a Graph
- Depth-First Traversal
- Breadth-First Traversal
- Dijkstra’s Algorithm for Shortest Path

TRAVERSING A GRAPH

- Suppose we had the following graph…



- We might assume we start at A
- How do we traverse it from there though?

TRAVERSING A GRAPH

- As this is a directed graph, we have three easy choices: D, F, and G

- Here's the difficult question – what do we do from there?

- Sometimes the choice is made for us (G must go to C)
- Other times our choice may be arbitrary (we might go in alphabetical order, or the order used in our adjacency list)

TRAVERSING A GRAPH

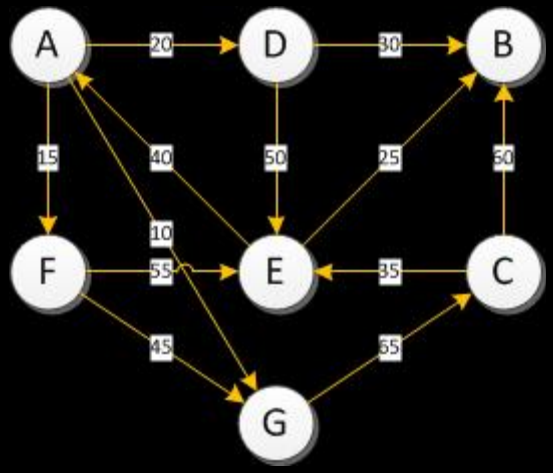
- Let's say we choose D – we must remember that F and G are on hold and also have to be traversed at some later point
- From D, we must choose B or E

- Suppose we choose B – again we must remember that E is on hold and we'll have to get back to it eventually

- B is a dead-end, so we can stop this part of our traversal here

TRAVERSING A GRAPH

- One method of traversing the remaining nodes is to backtrack to the vertex we came from and try the next adjacent vertex



- From B, we go back to D and then choose E
- Successive backtracking (sounds like recursion) should visit all our nodes right?

TRAVERSING A GRAPH

- Nope! There may be a complication with our plan
- When we traverse A D B E, in that order, we go back to A

- We've already been to A!!! This is a cycle, and they can be very dangerous - think infinite loop

- Therefore, we need some way to mark the vertex as it is visited so we do not visit it twice

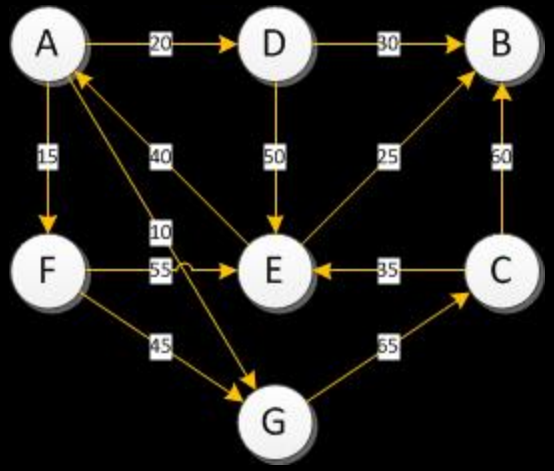
TRAVERSING A GRAPH

- We'll use the colour field of our GVertex structure to denote whether a node has been visited or not
- A colour gives a little more flexibility than a true/false value
- We will set the vertex to white before it is visited and black after it is visited
- For some algorithms, it may also be set to gray while it is being examined for edges, etc. Just know that a non-white value means the vertex has been seen

DEPTH-FIRST

1. Start at A
2. Follow A to D
3. Follow D to B
4. Backtrack to D and visit E
5. Follow E to A, but A is black
6. Backtrack to D, backtrack to A
7. Follow A to F
8. Follow F to E, but E is black
9. Follow F to G
10. Follow G to C
11. Follow C to B, but B is black
12. Follow C to E, but E is black
13. Backtrack to G, backtrack to F, backtrack to A
14. Follow A to G, but G is black – done!

A D B E F G C



DEPTH FIRST

- We have performed what is called a depth-first traversal of our graph
- Essentially, we went as deep into the graph as we could go, backtracked a level, went as deep as possible again, repeat
- A depth-first traversal of a tree defines a depth-first tree (or more commonly a forest!)

DEPTH FIRST

- A forest? Why?

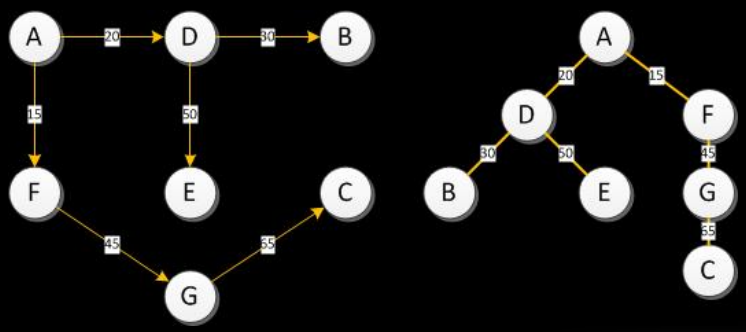
- What happens if we start at B? We cannot proceed any further
- Similarly, what if there is a vertex with no incoming edges? We may never visit it!

- The usual way to ensure every vertex is visited at least once is to do a depth-first traversal, marking each visited node accordingly, for every vertex in the graph (only mark each vertex once)

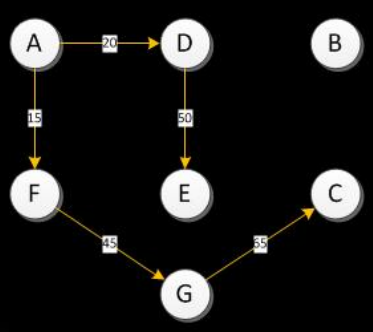
DEPTH FIRST

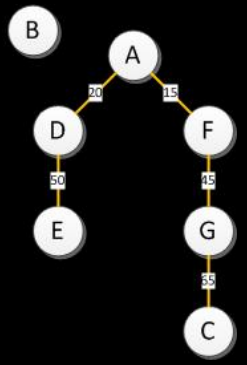
- Is it really a tree/forest though? Consider these two possible traversals of our graph – Wow!!!

Starting at A



Starting at B





DEPTH-FIRST PSEUDO-CODE

- Following this pseudo-code algorithm will traverse all the nodes from vertex v using depth-first

```
void dfTraverse(v) {
        print v.id
        v.colour = gray
        for each edge in v as x
                if (x.colour == white) dfTraverse(x)
        v.colour = black
}
```

DEPTH-FIRST PSEUDO-CODE

- Don’t forget that we have to run dfTraverse for every vertex though too! We must be sure we went everywhere – even unattached or unreachable vertices!

```
void depthFirstTraverse(Graph) {
        for each vertex v in Graph
                v.colour = white
        for each vertex v in Graph
                if (v.colour == white) dfTraverse(v)
}
```

DEPTH-FIRST TIPS

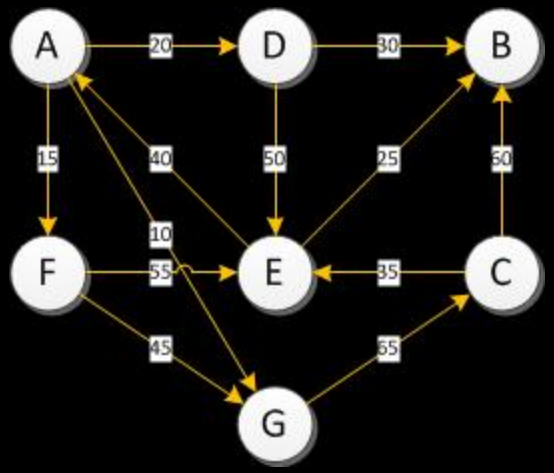
- One application of a depth-first is topological sorting. Suppose we can only complete some tasks after other have been complete
  - Building a car: the engine cannot be assembled before the parts that go inside it
  - Taking courses at college: we learn how to program in Java before we get really good at it in C here, etc.
- Setting up a graph with the requirements and performing a depth-first search is one way to solve these problems
- Note: the weights of the edges can be used to help pick one path over another (more on this later)
- Just be careful your tree does not go too deep – some trees are very large!

BREADTH-FIRST

- The other popular way to traverse a graph is breadth-first
- It is very similar to level-order traversal for a tree

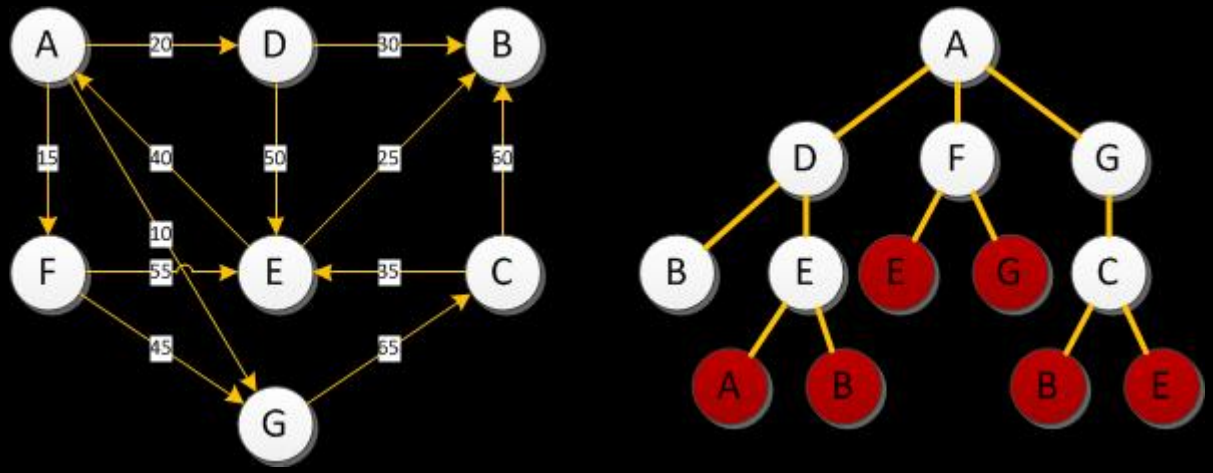
1. Start at A
2. Visit D, F, G in that order
3. From D, visit B, E in that order
4. From F, we have visited the children
5. From G, visit C
6. From B, we have visited the children
7. From E, we have visited the children
8. From C, we have visited the children

A D F G B E C



BREADTH-FIRST

- Let's convert that graph to our breadth-first tree to make it a little more obvious



- The red nodes in the tree are only present to illustrate the searching process – they have already been visited

BREADTH-FIRST PSEUDO-CODE

- Interestingly, we can perform a breadth-first traversal using a queue

```
Initialize a queue, Q
Set the colour of all vertices to white
Enqueue the start vertex, s
s.colour = gray
while (Q is not empty)
        p = Q.dequeue
        visit p
        for each edge of p as x
                if (x.colour is white) {
                        x.colour = gray
                        enqueue x
                }
        }
        p.colour = black
}
```

Questions?

DIJKSTRA'S ALGORITHM

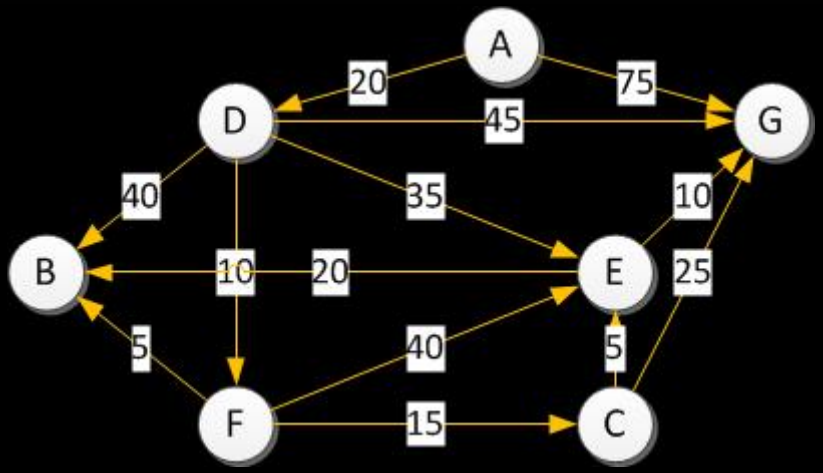
- Here is where the weights come in!

- Dijkstra's algorithm is an optimal method for finding the shortest path from source node S to destination node D where "shortest" is defined as the lowest sum of weights of the edges on a path from S to D

- This is sometimes referred to as the least-cost (for practical reasons) or minimum-weight path (more correctly)

DIJKSTRA'S ALGORITHM

- Suppose we have the following graph



- We might want to find the shortest path from A to E
- As a quick exercise, name some paths from A to E! Which do you think is best?

DIJKSTRA'S ALGORITHM

- We might consider A -> D -> E
  - With a cost of 20 + 35 = 55
- We might also consider A -> D -> F -> E
  - With a cost of 20 + 10 + 40 = 70
- Finally, we might consider A -> D -> F -> C -> E
  - With the smallest cost of 20 + 10 + 15 + 5 = 50!!!

DIJKSTRA'S ALGORITHM

- It turns out to solve this problem we must find the shortest paths from the source to all other vertices

- A Dutch mathematician named E. W. Dijkstra received the Turing Award in 1972 for his "fundamental contributions to developing programming languages"

- His algorithm was conceived in 1956 and published in 1959 – he was lucky enough to see his work make a major contribution to computer science before passing away in 2002

DIJKSTRA'S ALGORITHM

- Begin with all the vertices in a queue
- Initially set all the shortest path parents of each node to none, and all the costs to infinity (a rogue value)

vertex	A	B	C	D	E	F	G
parent	Nil	Nil	Nil	Nil	Nil	Nil	Nil
cost	0	infinity	infinity	infinity	infinity	infinity	infinity

- The idea is that eventually, we will know the cheapest parent for every vertex by only updating values where the total is smaller
- We'll start with A at a cost of 0 as we're already there

DIJKSTRA'S ALGORITHM

- We will place the lowest cost vertices in order at each step
- We begin with A, set it as visited, and look at its edges
- Turns out A goes to D (20) and G (75)
- 20 and 75 are both less than the previous infinity, so our table becomes …

vertex	A	B	C	D	E	F	G
parent	Nil	Nil	Nil	A	Nil	Nil	A
cost	0	infinity	infinity	20	infinity	infinity	75

DIJKSTRA'S ALGORITHM

- We continue with the lowest vertex D
- So let's visit D and process the cost. The trick is to process the cost to leave its edges plus its own cost (already determined)!
- D goes to B (40 + 20 = 60), which is lower than B currently (infinity) so we update B. D also goes to E (35 + 20 = 55), and F (10 + 20 = 30), etc.
- When we get to G, we discover that the cost is less (45 + 20 = 65) than the old value of 75, so we update G again to parent D and the new cost

vertex	A	B	C	D	E	F	G
parent	Nil	D	Nil	A	D	D	D
cost	0	60	infinity	20	55	30	65

DIJKSTRA'S ALGORITHM

- F is our new lowest cost vertex
- Therefore, visit F (at a cost of 30) and see it goes to B (5 + 30 = 35) at a lower cost, so we update B
- We see F goes to C (15 + 30 = 45) at a lower cost than infinity, so we update C
- F goes to E (40 + 30 = 70), but at a higher cost, so we ignore

vertex	A	B	C	D	E	F	G
parent	Nil	F	F	A	D	D	D
cost	0	35	45	20	55	30	65

DIJKSTRA'S ALGORITHM

- The new lowest unvisited value is B
- We examine B, but there are no edges leaving it – simple.
- Next lowest unvisited is C, and C goes to E (5 + 45 = 50), which is lower than E's current cost of 55, so we update
- C goes to G (25 + 45 = 70) which is higher, so we ignore

vertex	A	B	C	D	E	F	G
parent	Nil	F	F	A	C	D	D
cost	0	35	45	20	50	30	65

DIJKSTRA'S ALGORITHM

- Two nodes to go (E and G)!
- E is the lowest unvisited, so we look at it and see it goes to G (10 + 50 = 60), which is lower, so we update
- Finally, G
- We examine G and see it has no edges - we're done!

vertex	A	B	C	D	E	F	G
parent	Nil	F	F	A	C	D	E
cost	0	35	45	20	50	30	60

DIJKSTRA'S ALGORITHM

vertex	A	B	C	D	E	F	G
parent	Nil	F	F	A	C	D	E
cost	0	35	45	20	50	30	60

- Cost to B: 35, Path A -> D -> F -> B
- Cost to C: 45, Path A -> D -> F -> C
- Cost to D: 20, Path A -> D
- Cost to E: 50, Path A -> D -> F -> C -> E
- Cost to F: 30, Path A -> D -> F
- Cost to G: 60, Path A -> D -> F -> C -> E -> G

DIJKSTRA'S ALGORITHM

- The path to any vertex can be found by following the parent pointers
- For example, the parent of B is F, the parent of F is D, and the parent of D is A
- Therefore, the path from A to B is A -> D -> F -> B
- What is the path from E to A?
- The code is available in SLATE – have a peek

Questions?