ANALYSIS OF DIJKSTRA’S ALGORITHM
The initialization requires $\Theta(|V|)$ memory and run time.

We iterate $|V| - 1$ times, each time finding the next closest vertex to the source.

- Iterating through the table requires is $\Theta(|V|)$ time.
- Each time we find a vertex, we must check all of its neighbors.
- With an adjacency matrix, the run time is $\Theta(|V|(|V| + |V|)) = \Theta(|V|^2)$.
- With an adjacency list, the run time is $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$ as $|E| = O(|V|^2)$.

Can we do better?

- Recall, we only need the closest vertex.
- How about a priority queue?
Pseudocode (Using PQ)

```java
void dijkstra(s) {
    queue = new PriorityQueue<Vertex>();
    for (each vertex v) {
        v.dist = infinity;  // can use Integer.MAX_VALUE
        queue.enqueue(v);
        v.pred = null;
    }
    s.dist = 0;

    while (!queue.isEmpty()) {
        u = queue.extractMin();
        for (each vertex v adjacent to u)
            relax(u, v);
    }
}

void relax(u, v) {
    if (u.dist + w(u,v) < v.dist) {
        v.dist = u.dist + w(u,v);
        v.pred = u;
    }
}
```

Source: http://www.cs.dartmouth.edu/~thc/cs10/lectures/0509/0509.html
CSC 172, Fall 2017
The initialization still requires $\Theta(|V|)$ memory and run time
- The priority queue will also require $O(|V|)$ memory
- We must use an adjacency list, not an adjacency matrix

We iterate $|V| - 1$ times, each time finding the closest vertex to the source
- Place the distances into a priority queue
- The size of the priority queue is $O(|V|)$
- Thus, the work required for this is $O(|V| \ln(|V|))$

Is this all the work that is necessary?
- Recall that each edge visited may result in updating the priority queue
- Thus, the work required for this is $O(|E| \ln(|V|))$

Thus, the total run time is $O(|V| \ln(|V|) + |E| \ln(|V|)) = O(|E| \ln(|V|))$
Summary

We have seen an algorithm for finding single-source shortest paths

– Start with the initial vertex
– Continue finding the next vertex that is closest

Dijkstra’s algorithm always finds the next closest vertex

– It solves the problem in $O( |E| \ln(|V|))$ time using Priority Queue

– The algorithm can be further improved to run in time $O((|V|\ln(|V|) + |E|)$ --- (using Fibonacci Heap; We won’t cover it)
BACK TO GRAPH BASICS
A graph is a tree if it is connected and there is a unique path between any two vertices

- Three trees on the same eight vertices

- The graph is *acyclic*, that is, it does not contain any cycles
- Adding one more edge must create a cycle
- Removing any one edge creates two disjoint non-empty sub-graphs
Any tree can be converted into a **rooted tree** by:

– Choosing any vertex to be the root
– Defining its neighboring vertices as its children

and then recursively defining:

– All neighboring vertices other than that one designated its parent are now defined to be that vertices' children

Given this tree, here are three rooted trees (out of many others) associated with it
A forest is any graph that has no cycles.

Consequences:
- The number of edges is $|E| < |V|$
- The number of trees is $|V| - |E|$
- Removing any one edge adds one more tree to the forest

Here is a forest with 22 vertices and 18 edges
- There are four trees
In a *directed graph*, the edges on a graph are be associated with a direction

- Edges are ordered pairs \((v_j, v_k)\) denoting a connection from \(v_j\) to \(v_k\)
- The edge \((v_j, v_k)\) is **different** from the edge \((v_k, v_j)\)
Directed graphs

Given our graph of nine vertices \( V = \{v_1, v_2, \ldots, v_9\} \)

- These six pairs \((v_j, v_k)\) are directed edges
  \[ E = \{(v_1, v_2), (v_3, v_5), (v_5, v_3), (v_6, v_9), (v_8, v_4), (v_9, v_4)\}\]
The maximum number of directed edges in a directed graph is

\[ |E| \leq 2 \binom{|V|}{2} = 2 \frac{|V|(|V| - 1)}{2} = |V|(|V| - 1) = O\left(|V|^2\right) \]
A directed acyclic graph is a directed graph which has no cycles.

- These are commonly referred to as DAGs.
- They are graphical representations of partial orders on a finite number of elements.

These two are DAGs:

This directed graph is not acyclic:
GRAPH TRAVERSALS
We will look at traversals of graphs

- **Breadth-first** or **depth-first** traversals
- Must avoid **cycles**
- **Depth-first** traversals can be recursive or iterative
Strategies

Traversals of graphs are also called searches

We can use either breadth-first or depth-first traversals

– Breadth-first requires a queue
– Depth-first requires a stack

We each case, we will have to track which vertices have been visited requiring $\Theta(|V|)$ memory

– One option is a hash table

The time complexity cannot be better than and should not be worse than $\Theta(|V| + |E|)$

– Connected graphs simplify this to $\Theta(|E|)$
– Worst case: $\Theta(|V|^2)$
Breadth-first traversal

Consider implementing a breadth-first traversal on a graph:
– Choose any vertex, mark it as visited and push it onto queue
– While the queue is not empty:
  • Pop to top vertex $v$ from the queue
  • For each vertex adjacent to $v$ that has not been visited:
    – Mark it visited, and
    – Push it onto the queue

This continues until the queue is empty
– Note: if there are no unvisited vertices, the graph is connected,
Iterative Breadth-first traversal

An implementation can use a queue

Does this remind you of any other algorithm?

Level order traversal
Breadth-first traversal

The size of the queue is $O(|V|)$

– The size depends both on:
  • The number of edges, and
  • The out-degree of the vertices
Consider implementing a depth-first traversal on a graph:

– Choose any vertex, mark it as visited
– From that vertex:
  • If there is another adjacent vertex not yet visited, go to it
  • Otherwise, go back to the most previous vertex that has not yet had all of its adjacent vertices visited and continue from there
– Continue until no visited vertices have unvisited adjacent vertices

Two implementations:
– Recursive
– Iterative
Recursive depth-first traversal

A recursive implementation uses the call stack for memory:
Iterative depth-first traversal

An iterative implementation can use a stack

Does this remind you of any other algorithm?

Reverse Pre-order traversal
Example

Consider this graph
Example

Performing a **breadth-first** traversal
– Push the first vertex onto the queue
Performing a breadth-first traversal
– Pop A and push B, C and E

A
Example

Performing a breadth-first traversal:
– Pop B and push D
  A, B
Performing a breadth-first traversal:
– Pop C and push F
A, B, C
Example

Performing a breadth-first traversal:
– Pop E and push G and H
  A, B, C, E
Performing a breadth-first traversal:
– Pop D

A, B, C, E, D
Performing a breadth-first traversal:

- Pop F

A, B, C, E, D, F
Performing a breadth-first traversal:
– Pop G and push I

A, B, C, E, D, F, G
Performing a breadth-first traversal:

– Pop H

A, B, C, E, D, F, G, H
Performing a breadth-first traversal:

- Pop I

A, B, C, E, D, F, G, H, I
Performing a breadth-first traversal:
– The queue is empty: we are finished
  A, B, C, E, D, F, G, H, I
Example

Perform a recursive depth-first traversal on this same graph.
Performing a recursive depth-first traversal:
– Visit the first node

A
Example

Performing a recursive depth-first traversal:
– A has an unvisited neighbor
  
  A, B
Performing a recursive depth-first traversal:
– B has an unvisited neighbor
  A, B, C
Performing a recursive depth-first traversal:
– C has an unvisited neighbor
   A, B, C, D
Performing a recursive depth-first traversal:
– D has no unvisited neighbors, so we return to C
A, B, C, D, E
Performing a recursive depth-first traversal:
– E has an unvisited neighbor
  A, B, C, D, E, G
Performing a recursive depth-first traversal:

– F has an unvisited neighbor

A, B, C, D, E, G, I
Example

Performing a recursive depth-first traversal:
– H has an unvisited neighbor

A, B, C, D, E, G, I, H
Performing a recursive depth-first traversal:

– We recurse back to C which has an unvisited neighbour

A, B, C, D, E, G, I, H, F
Performing a recursive depth-first traversal:

- We recurse finding that no other nodes have unvisited neighbours

A, B, C, D, E, G, I, H, F
Performing a recursive depth-first traversal:
– We recurse finding that no other nodes have unvisited neighbours
  A, B, C, D, E, G, I, H, F
The order in which vertices appear can differ greatly
– An iterative depth-first traversal may also be different again

A, B, C, E, D, F, G, H, I

A, B, C, D, E, G, I, H, F
Applications of tree traversals include:

– Determining connectiveness and finding connected sub-graphs

– Determining the path length from one vertex to all others
This topic covered graph *traversals*
- Considered breadth-first and depth-first traversals
- Depth-first traversals can recursive or iterative
- More overhead than traversals of rooted trees
- Considered an example with both implementations
- They are also called *searches*
CONNECTEDNESS
Outline

We will use graph traversals to determine:

– Whether one vertex is connected to another
– The connected sub-graphs of a graph
First, let us determine whether one vertex is connected to another

- \( v_j \) is connected to \( v_k \) if there is a path from the first to the second

Strategy:

- Perform a breadth-first traversal starting at \( v_j \)
- While looping, if the vertex \( v_k \) ever found to be adjacent to the front of the queue, return true
- If the loop ends, return false
Consider implementing a breadth-first traversal on a graph:
– Choose any vertex, mark it as visited and push it onto queue
– While the queue is not empty:
  • Pop to top vertex \( v \) from the queue
  • For each vertex adjacent to \( v \) that has not been visited:
    – Mark it visited, and
    – Push it onto the queue

This continues until the queue is empty
– Note: if there are no unvisited vertices, the graph is connected,
Is A connected to D?
Determining Connections

Vertex A is marked as visited and pushed onto the queue.
Determining Connections

Pop the head, A, and mark and push B, F and G
Determining Connections

Pop B and mark and, in the left graph, mark and push H
- On the right graph, B has no unvisited adjacent vertices
Determining Connections

Popping F results in the pushing of E

![Diagram of connections between nodes A, B, C, D, E, F, G, and H.]
Determining Connections

In either graph, G has no adjacent vertices that are unvisited
Determining Connections

Popping H on the left graph results in C and I being pushed
Determining Connections

The queue on the right is empty

– We determine A is not connected to D
On the left, we pop C and return true because D is adjacent to C

– In the left graph, A is connected to D

```
    I
  E   F
    G
    H
    C
  A   B
    D
```
Determining Connections

On the left, we pop C and return true because D is adjacent to C

– In the left graph, A is connected to D
Summary

This topic covered connectedness
– Determining if two vertices are connected
PATH LENGTH
This topic looks at another problem solved by breadth-first traversals

– Finding all path lengths in an unweighted graph
Problem: find the distance from one vertex \( v \) to all other vertices

- Use a breadth-first traversal
- Vertices are added in *layers*
- The starting vertex is defined to be in the zeroeth layer, \( L_0 \)
- While the \( k \)th layer is not empty:
  
  - All unvisited vertices adjacent to vertices in \( L_k \) are added to the \((k + 1)\)st layer

Any unvisited vertices are said to be an infinite distance from \( v \)

Reference: Kleinberg and Tardos
Determining Distances

Consider this graph: find the distance from A to each other vertex
Determining Distances

A forms the zeroeth layer, $L_0$
Determining Distances

The unvisited vertices B, F and G are adjacent to A
– These form the first layer, \( L_1 \)
We now begin popping $L_1$ vertices: pop B
- H is adjacent to B
- It is tagged $L_2$
Determining Distances

Popping F pushes E onto the queue
– It is also tagged $L_2$
We pop G which has no other unvisited neighbours
– G is the last $L_1$ vertex; thus H and E form the second layer, $L_2$
Determining Distances

Popping H in $L_2$ adds C, I, and D to the third layer $L_3$
E has no more adjacent unvisited vertices

– Thus C, I, and D form the third layer, $L_3$
Determining Distances

Pop C, I, D. The queue is Empty. Stop.
This topic found the unweighted path length from a single vertex to all other vertices

– A breadth-first traversal was used
– The first vertex is marked as layer $0$
– Vertices added to the queue by one in layer $k$ are marked as layer $k + 1$
– Later, we will see different algorithms for finding the shortest path length in weighted graphs
TOPOLOGICAL SORTING
In this topic, we will discuss:

- Motivations
- Review the definition of a directed acyclic graph (DAG)
- Describe a topological sort and applications
Motivation

Given a set of tasks with dependencies, is there an order in which we can complete the tasks?
Restriction of paths in DAGs

In a DAG, given two different vertices $v_j$ and $v_k$, there cannot both be a path from $v_j$ to $v_k$ and a path from $v_k$ to $v_j$.

Proof:
- Assume otherwise; thus there exists two paths:
  $(v_j, v_{1,1}, v_{1,2}, v_{1,3}, \ldots, v_k)$
  $(v_k, v_{2,1}, v_{2,2}, v_{2,3}, \ldots, v_j)$
  From this, we can construct the path
  $(v_j, v_{1,1}, v_{1,2}, v_{1,3}, \ldots, v_k, v_{2,1}, v_{2,2}, v_{2,3}, \ldots, v_j)$
  This a path is a cycle, but this is an acyclic graph.
  $\therefore$ contradiction
Definition of topological sorting

A topological sorting of the vertices in a DAG is an ordering

\[ v_1, v_2, v_3, \ldots, v_{|V|} \]

such that \( v_j \) appears before \( v_k \) if there is a path from \( v_j \) to \( v_k \)
Definition of topological sorting

Given this DAG, a topological sort is

Example

For example, there are paths from H, C, I, D and J to F, so all these must come before F in a topological sort

\[ H, C, I, D, J, A, F, B, G, K, E, L \]

Clearly, this sorting need not be unique
Applications

Consider a gentleman is getting ready for a dinner out

He must wear the following:
  – jacket, shirt, briefs, socks, tie, etc.

There are certain constraints:
  – the pants really should go on after the briefs,
  – socks are put on before shoes
Otherwise

Gentleman to Superman

CSC 172, Fall 2017
Applications

The following is a task graph for getting dressed:

One topological sort is:
briefs, pants, wallet, keys, belt, socks, shoes, shirt, tie, jacket, iPod, watch

A more reasonable topological sort is:
b briefs, socks, pants, shirt, belt, tie, jacket, wallet, keys, iPod, watch, shoes
A graph is a DAG if and only if it has a topological sorting
Example

On this graph, iterate the following $|V| = 12$ times

- Choose a vertex $v$ that has in-degree zero
- Let $v$ be the next vertex in our topological sort
- Remove $v$ and all edges connected to it
Example

Let’s step through this algorithm with this example
- Which task can we start with?
Of Tasks C or H, choose Task C
Having completed Task C, which vertices have in-degree zero?
Only Task H can be completed, so we choose it.
Having removed H, what is next?

C, H
Both Tasks D and I have in-degree zero
– Let us choose Task D
Example

We remove Task D, and now?

C, H, D
Both Tasks A and I have in-degree zero
– Let’s choose Task A

C, H, D
Having removed A, what now?

C, H, D, A
Example

Both Tasks B and I have in-degree zero
– Choose Task B

C, H, D, A
Removing Task B, we note that Task E still has an in-degree of two.

- Next?

C, H, D, A, B
As only Task I has in-degree zero, we choose it

C, H, D, A, B
Example

Having completed Task I, what now?

C, H, D, A, B, I
Example

Only Task J has in-degree zero: choose it

C, H, D, A, B, I
Having completed Task J, what now?

C, H, D, A, B, I, J

C, H, D, A, B, I, J
Only Task F can be completed, so choose it
What choices do we have now?

C, H, D, A, B, I, J, F
Example

We can perform Tasks G or K
– Choose Task G

C, H, D, A, B, I, J, F
Example

Having removed Task G from the graph, what next?

C, H, D, A, B, I, J, F, G
Example

Choosing between Tasks E and K, choose Task E

C, H, D, A, B, I, J, F, G
At this point, Task K is the only one that can be run.

And now that both Tasks G and K are complete, we can complete Task L.
Example

There are no more vertices left

Thus, one possible topological sort would be:

Note that topological sorts need not be unique:

In this topic, we have discussed topological sorts
– Sorting of elements in a DAG
• A lot of these slides are taken from ECE 250 *Algorithms and Data Structures* course (University of Waterloo) offered by Prof. Douglas W. Harder.