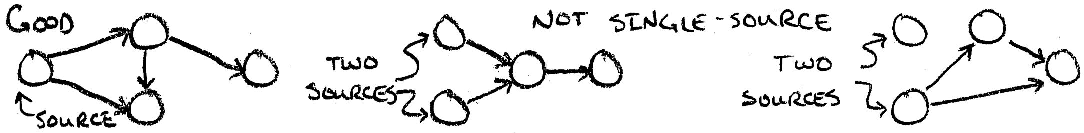
SINGLE - SOURCE SHORTEST PATHS

A path in a directed graph with only one starting node from which all other nodes in the graph can be found. If a node cannot be found from the source node than the graph is not a single source graph since even a disconnected node serves as its own source.



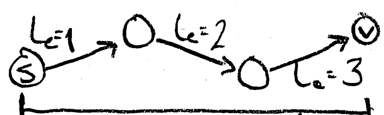
Dijkstra's shortest path algorithm works well with this type of problem.

INPUT: Directed graph, G, with vertices/nodes, V, and edges/relationships, E.

A relationship in E of nodes u and v denoted by (u, v) represents u serving as the tail or source and v sitting at the head or sink. $u \longrightarrow v$

Each edge has non-negative length, l_e.
There is always a single starting source vertex, s.

OUTPUT: For each node, v, in the set of vertices, V, compute L(v) which is the shortest path s–v in G.



$$L(v) = 6 = \sum_{i=s}^{v} l_{e_i}$$

ASSUMPTIONS: 1) All vertices are reachable by s, otherwise this would not be a single source problem. If not, run a breadth first search and remove all elements not connected to s.

2) $l_e \geq \emptyset$ for all edges

(con't on next sheet)

WHY ANOTHER SHORTEST PATH ALGORITHM?

Breadth First Search (BFS) can already compute shortest paths in linear time.

True, if $l_e = 1$ for every edge, e, in E, otherwise you will get an answer of how many nodes are between s and your destination vertex, not the actual distance.

Why not just replace each edge, e, by a directed path of l_e unit length edges? For example $\bullet \xrightarrow{3} \bullet = \bullet \xrightarrow{1} \bullet \xrightarrow{1} \bullet \xrightarrow{1} \bullet$

This is a clever simplification, but working with large graphs would blow up the graph to an even bigger problem significantly affecting runtime. Dijkstra's algorithm runs directly on the original directed graph.

PSEUDOCODE

INITIALIZE

```
1  X = [List of vertices processed so far from s to current vertex]
2  A[s] = Ø   # Array storing computed shortest path distance
              # from s to an explored vertex.
3  B[s] = s   # Array used to store the shortest path from s
              # to an explored vertex, that is, s -> a -> e, not
              # the distance. This array is only used to help
              # demonstrate the inner workings of the algorithm
              # and would not be used in production.
```

MAIN LOOP

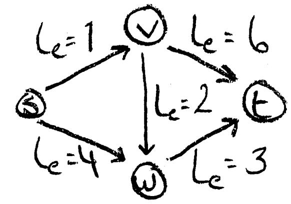
```
   # Let V represent all vertices in the graph
4  While X ≠ V:
      # Let v be an already explored node in X
      # Let w be an unexplored node in V, but not yet in X
5     For all edges (v*, w*): # v* = tail in X, w* = head not in X
6        Pick the one that minimizes A[v*] + l(v*,w*)  # Dijkstra's
                                                        # greedy criterion
7        Minimum found in line 6 = (v*, w*)
8     Add w* to X
9     Set A[w*] = A[v*] + l(v*, w*)

10    Set B[w*] = B[v*] + (v*, w*)
```

(con't on next sheet)

PSEUDOCODE EXAMPLE

GIVEN:



FIND: The shortest path of the graph using Dijkstra's Shortest path algorithm from s to t.

INITIALIZE: $X = [s]$ # Set of explored nodes
$A[s] = \emptyset$ # Shortest path length to explored node
$B[s] = s$ # Nodes in shortest path to an explored vertex

MAIN LOOP, $X \neq V$ # $X = [s]$, $V = [s, v, w, t]$
For all edges (v^*, w^*) # (s, v), (s, w)
Pick the one that minimizes $A[v^*] + L_{(v^*, w^*)}$ # Dijkstra's # greedy criterion

$A[s] + L_{(s,v)} = \emptyset + 1 = 1$
$A[s] + L_{(s,w)} = \emptyset + 4 = 4$
Use edge (s, v) # $1 < 4$
Add v to set X
Set $A[v] = A[s] + L_{(s,v)}$ # $A[v] = \emptyset + 1 = 1$
Set $B[v] = B[s] + v$ # $B[v] = (s) + v = s \to v$

MAIN LOOP, $X \neq V$ # $X = [s, v]$, $V = [s, v, w, t]$
For all edges (v^*, w^*) # (v, w), (v, t), (s, w)
Pick the one that minimizes $A[v^*] + L_{(v^*, w^*)}$
$A[v] + L_{(v,w)} = 1 + 2 = 3$
$A[v] + L_{(v,t)} = 1 + 6 = 7$
$A[s] + L_{(s,w)} = \emptyset + 4 = 4$
Use edge (v, w) # $3 < 4 < 7$
Add w to set X
Set $A[w] = A[v] + L_{(v,w)}$ # $A[w] = 1 + 3 = 4$
Set $B[w] = B[v] + w$ # $B[w] = (s \to v) + w = s \to v \to w$

MAIN LOOP $X \neq V$ # $X = [s, v, w]$ # $V = [s, v, w, t]$
For all edges (v^*, w^*) # (v, t), (w, t)
Pick the one that minimizes $A[v^*] + L_{(v^*, w^*)}$
$A[v] + L_{(v,t)} = 1 + 6 = 7$
$A[w] + L_{(w,t)} = 3 + 3 = 6$
Use edge (w, t) # $6 < 7$
Add t to set X
Set $A[t] = A[w] + L_{(w,t)}$ # $A[t] = 3 + 3 = 6$
Set $B[t] = B[w] + t$ # $B[t] = (s \to v \to w) + t = s \to v \to w \to t$

MAIN LOOP $X = V$

SHORTEST PATH $s \to t = A[t] = 6$

```python
# dijktstra_shortest_path.py
"""
An implementation of Dijktra's shortest-path algorithm.

The file contains an adjacency list representation of an undirected
weighted graph with 200 vertices labeled 1 to 200. Each row consists
of the node tuples that are adjacent to that particular vertex along
with the length of that edge. For example, the 6th row has 6 as the
first entry indicating that this row corresponds to the vertex
labeled 6. The next entry of this row "141,8200" indicates that
there is an edge between vertex 6 and vertex 141 that has length 8200.
The rest of the pairs of this row indicate the other vertices adjacent
to vertex 6 and the lengths of the corresponding edges.

Your task is to run Dijkstra's shortest-path algorithm on this graph,
using 1 (the first vertex) as the source vertex, and to compute the
shortest-path distances between 1 and every other vertex of the graph.
If there is no path between a vertex v and vertex 1, we'll define the
shortest-path distance between 1 and v to be 1000000.

You should report the shortest-path distances to the following ten
vertices, in order: 7,37,59,82,99,115,133,165,188,197. Enter the
shortest-path distances using the fields below for each of the
vertices.

IMPLEMENTATION NOTES: This graph is small enough that the
straightforward O(mn) time implementation of Dijkstra's algorithm
should work fine. OPTIONAL: For those of you seeking an additional
challenge, try implementing the heap-based version. Note this
requires a heap that supports deletions, and you'll probably need
to maintain some kind of mapping between vertices and their positions
in the heap.
"""

from collections import Counter
from collections import defaultdict

class Node:
    def __init__(self, name, connections):
        self.name = name
        self.connections = connections

def main():

    # nodes put in an exploration array
```

```python
    # Have an array storing the distance from 1 to an explored node
    # Return the distances for the nodes requested.
    source_node = 1
    graph = setup()

    # Breadth-first search to eliminate nodes not connected to source
    # If node not connected to source distance = 1000000
    connected_to_source = breadth_first_search(graph, source_node)
    for node in graph.keys():
        if node in connected_to_source:
            continue
        else:
            graph[source_node].append((node, 1000000))

    distance_to_source = find_shortest_path(graph, source_node)

    # Return the distance from node 1 to the following ten vertices, in
    # order: 7, 37, 59, 82, 99, 115, 165, 188, 197
    distance_to_node = [7, 37, 59, 82, 99, 115, 133, 165, 188, 197]
    output(distance_to_node, distance_to_source)

def setup():
    """
    Converts dijkstraData.txt into a python useable format.
    """

    node_edges = defaultdict(list)

    with open("dijkstraData.txt") as f:
        # Exclude the newline character, \n
        for line in f:
            # Remove the trailing tab and newline characters.
            # Otherwise, there are issues on import.
            line = line.rstrip('\t\n')
            line = line.split('\t')

            source_node = int(line[0])

            for val in line[1:]:
                node_edge = val.split(',')
                node_edges[source_node].append((int(node_edge[0]),
int(node_edge[1])))

    return node_edges

```

```
def breadth_first_search(graph, source_node):
    """
    Conducts a breadth-first search to determine connections to the
    source node.
    """

    # Set all nodes as unexplored.
    nodes = defaultdict(list)
    # Set the source node as explored
    nodes["Explored"].append(source_node)

    # Let q = queue data structure (First-in, First-out (FIFO))
    # initialized with the source node.
    q = [source_node]

    while len(q) != 0:
        v = q.pop(0)
        # Explore the different edges v possesses (v, w)
        for connection in graph[v]:
            w = connection[0]
            if w in nodes["Explored"]:
                continue
            else:
                nodes["Explored"].append(w)
                q.append(w)

    return nodes["Explored"]

def find_shortest_path(graph, source_node):
    """
    Finds the shortest path between two nodes using Dijkstra's shortest
    path algorithm.
    """

    # Shortest distance to source node is 0.
    distances_to_node = [None] * len(graph)
    distances_to_node[0] = 0

    list_of_vertices_processed = [source_node]

    while Counter(list_of_vertices_processed) != Counter(graph.keys()):
        # (source_node, ending_node, distance)
        shortest_path = (source_node, source_node, 10000000) # Use arbitrarily
large number
        for v_star in list_of_vertices_processed:
```

```
            for w_star in graph[v_star]:
                if w_star[0] in list_of_vertices_processed:
                    continue
                else:
                    path_length = distances_to_node[v_star-1] + w_star[1]
                    if path_length < shortest_path[2]:
                        shortest_path = (source_node, w_star[0], path_length)
        if shortest_path[1] != source_node:
            list_of_vertices_processed.append(shortest_path[1])
            distances_to_node[shortest_path[1]-1] = shortest_path[2]

    return distances_to_node

def output(nodes_in_question, distances):
    """
    Prints the distance from the source node to the node in question.
    """

    for node in nodes_in_question:
        print("From node 1 to node {} the distance is: {}.".format(node,
distances[node-1]))

if __name__ == "__main__":
    main()
```

```go
/*
An implementation of Dijkstra's shortest-path algorithm.

The file contains an adjacency list representation of an undirected
weighted graph with 200 vertices labeled 1 to 200. Each row consists
of the node tuples that are adjacent to that particular vertex along
with the length of that edge. For example, the 6th row has 6 as the
first entry indicating that this row corresponds to the vertex
labeled 6. The next entry of this row "141,8200" indicates that
there is an edge between vertex 6 and vertex 141 that has length 8200.
The rest of the pairs of this row indicate the other vertices adjacent
to vertex 6 and the lengths of the corresponding edges.

Your task is to run Dijkstra's shortest-path algorithm on this graph,
using 1 (the first vertex) as the source vertex, and to compute the
shortest-path distances between 1 and every other vertex of the graph.
If there is no path between a vertex v and vertex 1, we'll define the
shortest-path distance between 1 and v to be 1000000.

You should report the shortest-path distances to the following ten
vertices, in order: 7,37,59,82,99,115,133,165,188,197. Enter the
shortest-path distances using the fields below for each of the
vertices.

IMPLEMENTATION NOTES: This graph is small enough that the
straightforward O(mn) time implementation of Dijkstra's algorithm
should work fine. OPTIONAL: For those of you seeking an additional
challenge, try implementing the heap-based version. Note this
requires a heap that supports deletions, and you'll probably need
to maintain some kind of mapping between vertices and their positions
in the heap.
*/
package main

import (
	"bufio"
	"fmt"
	"os"
	"strconv"
	"strings"
	"unicode"
)

type connection struct {
	Sink, Distance int
```

```
}

type path struct {
    Head, Tail, Length int
}

func main() {

    sourceNode := 1
    graph := setup()

    // Breadth-first search to eliminate nodes not connected to source.
    // If node nod connected to source distance = 1000000
    connected := breadthFirstSearch(graph, sourceNode)
    for node := range graph {
        if _, ok := connected[node]; !ok {
            graph[sourceNode] = append(graph[sourceNode], connection{node,
1000000})
        }
    }

    distanceToSource := findShortestPath(graph, sourceNode)

    // Return the distance from the source node to the following ten
    // vertices, in order: 7, 37, 59, 82, 99, 115, 165, 188, 197.
    nodesInQuestion := []int{7, 37, 59, 82, 99, 115, 165, 188, 197}
    output := genOutput(nodesInQuestion, distanceToSource)
    for i, v := range output {
        fmt.Printf("The distance to node %d from the source node is %d.\n",
nodesInQuestion[i], v)
    }
}

// setup converts dijkstraData.txt into Go useable format.
func setup() map[int][]connection {

    nodeMap := make(map[int][]connection)

    f, err := os.Open("dijkstraData.txt")
    if err != nil {
        fmt.Println(err)
    }
    defer f.Close()

    scanner := bufio.NewScanner(f)
```

```
    scanner.Split(bufio.ScanLines)

    for scanner.Scan() {
        xLine := strings.Split(scanner.Text(), "\t")
        var head int

        for _, val := range xLine {
            // Get rid of the parentheses
            val = strings.TrimFunc(val, func(r rune) bool {
                return !unicode.IsNumber(r)
            })

            splitVal := strings.Split(val, ",")

            var tail, distance int
            if len(splitVal) < 2 && splitVal[0] != "" {
                h, err := strconv.Atoi(splitVal[0])
                if err != nil {
                    fmt.Println(err)
                }
                head = h
            } else {
                for index, v := range splitVal {
                    if v != "" && index < 1 {
                        t, err := strconv.Atoi(v)
                        if err != nil {
                            fmt.Println(err)
                        }
                        tail = t
                    } else if v != "" && index > 0 {
                        d, err := strconv.Atoi(v)
                        if err != nil {
                            fmt.Println(err)
                        }
                        distance = d
                    }
                }
            }
            node := connection{
                tail,
                distance,
            }
            if node.Sink != 0 {
                nodeMap[head] = append(nodeMap[head], node)
            }
```

```
        }
    }
    return nodeMap
}

// breadthFirstSearch explores the given graph for a connection between
// the source node and all other nodes in the graph returning a new
// graph only with nodes connected to the source node.
func breadthFirstSearch(graph map[int][]connection, sourceNode int)
map[int][]connection {

    searchedGraph := make(map[int][]connection)
    for k, v := range graph {
        searchedGraph[k] = v
    }

    // Set all nodes as unexplored.
    isExplored := make(map[int]bool)
    for key := range graph {
        isExplored[key] = false
    }
    // Except the starting node.
    isExplored[sourceNode] = true

    // Let q = queue data structure (First-in, First-out (FIFO))
    // initialized with the source node.
    q := []int{sourceNode}

    for len(q) != 0 {
        u := q[0]
        if len(q) < 2 {
            q = nil
        } else {
            q = q[1:]
        }
        // Explore the different edges u possesses, (u, v).
        if _, prs := graph[u]; prs {
            for _, v := range graph[u] {
                if !isExplored[v.Sink] {
                    isExplored[v.Sink] = true
                    q = append(q, v.Sink)
                }
            }
        }
    }
```

```
    for node := range graph {
        if !isExplored[node] {
            delete(searchedGraph, node)
        }
    }
    return searchedGraph
}

// findShortestPath finds the shortest path using Dijkstra's shortest
// path algorithm from the source node to the node in question.
func findShortestPath(graph map[int][]connection, sourceNode int) []int {

    distanceTo := make([]int, len(graph))
    distanceTo[0] = 0

    // Let v be the list of vertices processed so far
    v := make(map[int]struct{})
    v[sourceNode] = struct{}{}

    for len(v) < len(graph) {
        // Use arbitrarily large number
        shortestPath := path{
            Head:   sourceNode,
            Tail:   sourceNode,
            Length: 10000000,
        }
        for vStar := range v {
            for _, wStar := range graph[vStar] {
                if _, prs := v[wStar.Sink]; !prs {
                    pathLength := distanceTo[vStar-1] + wStar.Distance
                    if pathLength < shortestPath.Length {
                        shortestPath = path{
                            Head:   sourceNode,
                            Tail:   wStar.Sink,
                            Length: pathLength,
                        }
                    }
                }
            }
        }
        if shortestPath.Tail != sourceNode {
            v[shortestPath.Tail] = struct{}{}
            distanceTo[shortestPath.Tail-1] = shortestPath.Length
        }
    }
```

```
    return distanceTo
}

// genOutput generates the output strings.
func genOutput(nodes, distance []int) []int {

    answers := make([]int, len(nodes))

    for idx, node := range nodes {
        answers[idx] = distance[node-1]
    }
    return answers
}

```