Lesson 11 Algorithms For Weighted Graphs: Creative Intelligence Manifesting As Material Creation **Wholeness of the Lesson** Weighted graphs are graphs that have weights or costs associated with each edge. Two questions that often need to be answered when working with weighted graphs are (1) What is the least costly path between two given vertices of the graph? (2) What is the least costly subgraph of the given graph which includes all the vertices of the given graph? Dijkstra’s Shortest Path Algorithm provides an efficient solution to the first

B C question; Kruskal’s Minimum Spanning Tree Algorithm provides an efficient solution to the second. Solutions to optimization problems of all kinds give expression to Nature’s tendency to achieve the most possible with the least expenditure of energy. Waking up to one’s own deeper values of intelligence has the effect of drawing Nature’s style of functioning into our thinking and action so that we automatically achieve goals with less effort.

0 8

A 4 8 7 22 1 3 2 5 E

D

3 9

F

85

1

Outline

Weighted graphs

Shortest path problem

’s Dijkstra

algorithm Minimum spanning tree problem

Kruskal

’s Algorithm

Shortest Paths 2

Weighted Graphs

In a weighted graph, each edge has an associated numerical

∎ In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports

SFOORD PVD

LAX

DFW

MIA LGA HNL

value, called the weight of the edge (wt: edges → numbers) Edge weights may represent distances, costs, etc.

Example:

Shortest Paths 3

Shortest Path Problem

Given a connected weighted graph and two vertices

∎ Internet packet routing ∎ Flight reservations ∎ Driving directions

SFOORD PVD

LAX

DFW

MIA LGA HNL

***s*** and ***x***, we want to find a path of minimum total weight between ***s*** and ***x.***

∎ "Length" of a path is the sum of the weights of its edges. Example:

∎ Shortest path between Providence and Honolulu Applications

Shortest Paths 4

Dijkstra’s Algorithm: The Problem

The

distance of a vertex ***v*** from a vertex ***s***, denoted ***d*(*s*,*v*)**, is the length of a shortest path between ***s*** and ***v***

***Question:***

Is it always true in a weighted graph that, for any two vertices *v, w*, d(*v,w*) = wt(*v,w*)? Prove or give a counterexample.

Dijkstra

’s algorithm computes the distances of all the vertices from a given start vertex ***s***

Assumptions:

∎ the graph G = (V,E) is connected ∎ the edges are undirected ∎ the edge weights are nonnegative

Shortest Paths 5

Starting with weighted graph G = (V,E) and starting vertex s, we

wish to compute, for each vertex v, the shortest distance from s to v in G.

We will store our computed value of the distance from s to any

vertex v in a map A: A[v] = our computed value of distance from s to v

If our algorithm is right (which we will need to prove) then, after

the algorithm has completed execution, we will have A[v] = d(s,v), for each v in V.

Shortest Paths 6

The Basic Idea

Basic Strategy:

It is reasonable to attempt to compute distances first for vertices close to s, then for vertices farther away. Step 1. We set A[s] = 0 (since d(

s,s) = 0). Step 2. Pick a vertex v adjacent to s so that (

s,v) has the least weight among all edges incident to s. ∎ It will turn out that for this v, a shortest path from s to v

really is wt(s,v). We will set A[v] = wt(s,v). ∎ However, it will not be true in general that for any other w

adjacent to s, wt(s,w) is a shortest path from s to w

Shortest Paths 7

Example

Step 2: Setting

The Logic: Why does Step 2 work? (Pick a vertex u adjacent to s so that (s,u) has least weight among edges incident to s.)

If p : s, w1, w2, ..., wk = u is any other path from s to u, then w1 must itself be u (otherwise the first part of the path – s, w1 – is already at least long as s, u)

A[u] = wt(s,u) = 1 is correct ∎ (s,u) has least weight

among all pairs (s,t) where t adjacent to s However, setting

A[v] = 5 is not correct

∎ A less costly path from s to v

than s,v is s, u, z, x, v

Basic Idea, continued

Step 3. After an optimal vertex v adjacent to

s has been chosen, there are two possible ways to extend further

∎ For some other vertex w adjacent to s, wt(s,w)

may turn out to be the shortest path length from s to w, OR ∎ For some w adjacent to v, the path s, v, w is

shortest possible from s to w. ∎ Among these choices, the algorithm will pick the

path that has minimal total weight

Shortest Paths 9

Example of Step 3

**Why Step 3 Works** Case I. wt(s,v) is minimum

Any path from s to v begins either as s, u or s, v. By assumption s, v is no longer than s, u, v. The other possibility is s, u, z, x, v, but s, v is (by assumption) no longer than s, u, z. Case II. A[u] + wt(u,v) is minimum Case III. A[u] + wt(u,z) is minimum.

Cases II and III are similar. We prove it for Case III: We want to show that s, u, z is a shortest path to z. The other possible choices are s, u, v, x, z and s, v, x, z Path s, u, v, x, z:

length s,u,z ≤ length of s,u,v (by assumption) Path s, v, x, z:

length of s, u, z ≤ length s, v (by assumption)

Shortest Paths 10

Dijkstra’s Algorithm

We extend Steps 1, 2, 3 till every vertex has been reached

***s*** In general, we build a "cloud" X of vertices, beginning with

and eventually including all of V. Associated with each vertex v in X will be a value A[v] representing the algorithm’s current estimate of the shortest path length from s to v

In each step, we add one new vertex w to X When w is placed in X (as we will show), the value stored in A[w] is precisely the value d(s, w).

Shortest Paths 11

Slow Dijkstra’s Algorithm

**Input:** A simple connected undirected weighted graph G with nonnegative edge weights, determined by a weight function wt (x,y), and a starting vertex s of G.

**Output:** Table A of shortest distances d(s,v) from s to v, for each v in V, so A[v] = d(s,v) for each v **Aux Output:** Table B with property that B[v] is a shortest path from s to v.

**The Algorithm:**

A [s] ← 0. X ← {s} //Basis step

**while** X ≠ V **do**

{ POOL ← {(v,w) ∈ E | v ∈ X and w ∉ X} } //here, we have no control over how much of E is searched, leading to slow running time (v’,w’) ← search POOL for edge (v,w) for which greedy length A[v] + wt(v,w) is minimal A[w’] ← A[v’] + wt(v’,w’) add w’ to X

Shortest Paths 12

Worked Example: Step 1

Shortest Paths 13

Worked Example: Step 2

Auxiliary Storage: B[v] = B[s] U {(s,v)} = {(s,v)}

Shortest Paths 14

Worked Example: Step 3

Auxiliary Storage: B[w] = B[v] U {(v,w)} = {(s,v), (v,w)}

Shortest Paths 15

Worked Example: Step 4

Computeed values of table B: B[s] = { }, B[v] = {(s,v)}, B[w] = {(s,v), (v,w)}, B[x] = {(s,v), (v,w), (w,x)}

Shortest Paths 16

Slow Dijkstra – Running Time ◆Running time without optimizations can be computed by observing that a (potentially) exhaustive search of edges is made in each iteration, leading to a running time of O(mn).

◆This can be improved to O(m \* log n) if an optimal

data structure is used.Shortest Paths 17

Improving Dijkstra

◆ Since "mins" are needed in each iteration, serve them using a

Priority Queue instead of doing an exhaustive search of edges. ◆ Begin by setting A[v] = ∞ for each v (other than starting vertex s) and progressively refine the value in A[v] as the algorithm proceeds. ◆ Use the values A[v] as keys in the priority queue, with

corresponding value v. At each step, the min in the priority queue represents the smallest among all approximations to a correct distance.

Shortest Paths 18

Dijkstra – Using Priority Queue

**Input:** Weighted undirected graph G=(V,E), starting vertex s

**Output:** Map A where for each vertex v, A[v] = d(s,v)

A[s] ← 0

A[v] ← ∞ (for each vertex v in V where v != s)

Q ← new heap-based priority queue **//each node in Q has value a vertex u and key A[u]**

while !Q.isEmpty() do

(u, A[u]) ← Q.removeMin() **//logically, put u in X; A[u] is now correct**

for each v in Q that is adjacent to u do

greedyLen ← A[u] + wt(u, v)

if greedyLen < A[v]

A[v] ← greedyLen

Q.updateNode(v, greedyLen) **//update value A[v] for v**

return the map A

Correctness

**Main Idea.** First s is brought into X and A[s] is correct. Then A[v] is updated for each v adjacent to s, and the smallest such A[v] becomes the min in the queue. Our earlier argument showed that in this case A[v] is correct and v is correctly pulled into X. Then the value A[w] is updated for every w adjacent to v, and again the min is chosen. We show that, for each such minimum element (v, A[v]), we have A[v] = d(s,v).

Shortest Paths 20

Main Lemma

Shortest Paths 21

Correctness Proof (1)

Assume that through the ith iteration, for each vertex v in X (i.e. removed from Q), we have A[v]=d(s,v). We show this holds through the i+1st iteration. Suppose at this stage that w is pulled off the Q (so enters X). We wish to show A[w] = d(s,w); assume that this is not the case, so that A[w] > d(s,w). We will arrive at a contradiction.

Let q: s, . . ., y, z, . . ., w be a true shortest path from s to w; let L denote the length of q. As in previous proof, z is the first vertex in q that is not yet in X. Let q0 : s, . . . , y, z be the subpath of q terminating in z; let L0 be the length of q0. We will show that A[w] ≤ L0 (and this will give us a contradiction).

Shortest Paths 22

Correctness Proof (2)

**Claim.** A[z] = d(s,z) **Proof.** Recall that earlier in the algorithm, when y was removed from Q, since z is adjacent to y, A[z] was possibly updated (after which A[z] = A[y] + wt(y,z)), and could have been updated again after that. It follows that

A[z] ≤ A[y] + wt(y,z) By assumption, since y is in X, A[y]=d(s,y). Also, since q0 is a true shortest path to z, by the Lemma, d(s,z) = d(s,y) + wt(y,z).

We haveA[z] ≤ A[y] + wt(y,z) = d(s,y) + wt(y,z) = d(s,z) However, for all u, A[u] ≥ d(s,u) (approximations decrease toward the true distance). It follows that

A[z] = d(s,z)

Shortest Paths 23

Correctness Proof (3)

**Continuation of the Main Proof**. Notice that in the i+1st stage, when w is being removed from Q, z is not yet in X, and (w, A[w]) is the minimum element of Q, and so A[w] ≤ A[z]. Therefore, by the Claim,

A[w] ≤ A[z] = d(s,z) = L0 ≤ L, as required.

Shortest Paths 24

Implementation Issues

How

can we locate a node in the queue that contains a vertex which is adjacent to a given vertex v? And how can we update such a node?

Solution

: Use auxiliary map M that matches vertices to nodes in the Queue. Given a vertex v, to locate node that contains a vertex adjacent to v, check adjacency list for next adjacent vertex u, then look up node n that contains u by consulting M. (This requires us to expose the private nodes of Q to the algorithm.)

To update the key (u, A[u]) in the queue, we enhance the priority queue operations to include

updateNode(n, key, value) (where n is a node in Q) which behaves as follows:

1. The current key in n is changed to ABSOLUTE\_MIN, and upheap

is performed (O(log n) time) 2. Perform removeMin to remove n from Q (O(log n) time) 3. Insert the new (key,value) using insertItem (O(log n) time)

Shortest Paths 25

Dijkstra – Using Priority Queue and Node Map

**Input:** Weighted undirected graph G=(V,E), starting vertex s

**Output:** Map A where for each vertex v, A[v] = d(s,v)

A[s] ← 0

A[v] ← ∞ (for each vertex v in V where v != s)

Q ← new heap-based priority queue **//each node in Q has value a vertex u and key A[u]**

M ← new HashMap {key/value pair is (v,n), v a vertex, n a node in heap containing v }

while !Q.isEmpty() do

(u, A[u]) ← Q.removeMin() **//logically, put u in X; A[u] is now correct**

M.remove(u)

for each v in Q that is adjacent to u do **//update values A[v] for v adjacent to u**

greedyLen= A[u] + wt(u, v)

if greedyLen < A[v]

n ← M.get(v)

A[v] ← greedyLen

Q.updateNode(n, v, greedyLen) **//update value A[v] for v if better value has been found**

return the map A

Worked Example (Start)

Shortest Paths 27

X = {}

Worked Example (Step 1)

(s,0) ← Q. removeMin M.remove(s) Vertices adjacent to s: v, w

Process v:

greedyLen = A[s] + wt(s,v) = 0 + 1 = 1 greedyLen < A[v] ? Yes A[v] ← 1 Q.updateNode(B, v, 1) Process w:

greedyLen = A[s] + wt(s,w) = 0 + 4 = 4 greedyLen < A[w] ? Yes A[w] ← 4 Q.updateNode(C, w, 4)

Shortest Paths 28

X = {s}

Worked Example (Step 2)

(v,1) ← Q. removeMin M.remove(v) Vertices adjacent to v still in Q: x, w

Process x:

greedyLen = A[v] + wt(v,x) = 1 + 6 = 7 greedyLen < A[x] ? Yes A[x] ← 7 Q.updateNode(D, x, 7) Process w:

greedyLen = A[v] + wt(v,w) = 1 + 2 = 3 greedyLen < A[w] ? Yes A[w] ← 3 Q.updateNode(C, w, 3)

Shortest Paths 29

X = {s, v}

Worked Example (Step 3)

(w,3) ← Q. removeMin M.remove(w) Vertices adjacent to w still in Q: x

Process x:

greedyLen = A[w] + wt(w,x) = 3 + 3 = 6 greedyLen < A[x] ? Yes A[x] ← 6 Q.updateNode(D, x, 6) Final Step:

(x,6) ← Q. removeMin M.remove(x) return A

Shortest Paths 30

X = {s, v, w}

// X = {s, v, w, x}

Running time

◆ Initialize A[v] for all vertices. O(n) ◆ Build priority queue for all vertices O(nlog n) ◆ Initialize the hashmap M O(n) ◆ The while loop removes one min node each time until the priority queue is

empty. So the algorithm is going to execute while loop n times.

◆ Remove min node and do downheap O(nlog n) ◆ Remove this node from the map M (O(1) each time) O(n) ◆ For each vertex v removed from queue and each w adjacent

to v in queue,

- update w in the queue (if necessary) O(mlog n)

[requires log(n) for each update, O(deg(v)) times =

]

◆ Since the graph is connected, n is O(m).

\_\_\_\_\_\_\_\_\_ **Total Running Time:** O(m log n)

(Note: This improves the O(mn) running time of the slow algorithm.) 31

Dijkstra - Exercises

Why is there a

requirement that edges have non- negative weights? Why can’t we add a large positive constant to every edge (to eliminate negative edge weights) and compute shortest paths for the new graph using Dijkstra?

Shortest Paths 32

Dijkstra - Exercises

Why is there a

requirement that edges have non- negative weights? Why can’t we add a large positive constant to every edge (to eliminate negative edge weights) and compute shortest paths for the new graph using Dijkstra?

Shortest Paths 33

Exercises, continued

Does Dijkstra’s Algorithm sometimes work correctly when there are negative edge weights? Consider this weighted graph.

Shortest Paths 34

Exercises, continued

Why is Dijkstra’s approach to the shortest path problem better than simply using BFS, as described in the previous lesson?

[BFS approach: Making all edge weights = 1 is same as removing all weights. Perform BFS with start vertex s and compute distance to each vertex by returning its level in the BFS spanning tree. These computed values should be same as values found using Dijkstra]Shortest Paths 35

Main Point

Dijkstra’s algorithm is an example of a shortest-path algorithm – an algorithm that efficiently (O(mlog n)) computes the shortest distance between two vertices in a graph. Analogously, Nature itself is known to obey the law of least action – Nature does the least possible amount of work to proceed from one location or state to another. Nature’s way of achieving this makes use of computational dynamics that involve “no effort” and no steps.

Shortest Paths 36

Minimum Spanning Tree Problem

Spanning subgraph

∎ Subgraph of a graph ***G***

containing all the vertices of ***G*** Spanning tree

∎ Spanning subgraph that is

itself a tree Minimum spanning tree (MST)

∎ Spanning tree of a weighted

graph with minimum total edge weight Applications

∎ Computer network (minimize

cost of cable) ∎ Transportation networks

(minimize cost of road construction)

ORD

10 1PIT DEN

6 93

8

5 2 DFW

ATL

7

4

STL

DCA

Shortest Paths 37

Kruskal's Greedy Strategy

Build a collection T of edges by doing the following: At each step, add an edge e to T of least weight subject to the constraint that adding e to T does not create a cycle in T. To answer questions about correctness and running time of this algorithm, we need to specify certain details.

Shortest Paths 38

Implementation Questions

1. How do we pick the next edge of least

weight at each step?

2. How do we make sure that we do not add

an edge to T that produces a cycle?

Shortest Paths 39

Solutions

1. We can arrange edges by sorting them by

weight (in ascending order), and so we pick edges according to this sorted order. (This is a greedy strategy.)

2. We can ensure no cycles are created by

building local minimum spanning trees around each vertex (to be explained further in coming slides)

Shortest Paths 40

Kruskal's Algorithm

First step is to sort all edges by weight.

clusters Second step involves creation of

∎ Every vertex is initially placed in a trivial cluster --

the cluster for a vertex v, denoted C(v), is simply {v}. A cluster represents a local minimum spanning tree. ∎ When the next edge (u,v) is considered, C(u) and C(v) are compared -- if different, (u,v) is included as an edge in the final output tree, and C(u) and C(v) are merged.

Shortest Paths 41

Kruskal’s Algorithm

**Input:** A simple connected weighted graph G = (V, E) with n vertices and m edges **Output:** A minimum spanning tree T of G

**The Algorithm:**

sort E in increasing order of edge weight

for each vertex *v* in *G*, define an elementary cluster *C*(*v*) (which will grow) by *C* (*v*) = {*v*}

T ← an empty tree //T will eventually become the minimum spanning tree

**while** T has fewer than n – 1 edges **do**

(u,v) ← next edge

C(v) ← cluster containing v

C(u) ← cluster containing u

**if** C(v) ≠ C(u) **then**

add edge (u,v) to T merge C(u) and C(v) (and update other clusters as needed)

**return** T

Note: We represent T as a collection of edges; to make it a graph, include as its vertex set all endpoints of the edges in T.

Shortest Paths 42

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {...}

**1**

**A2**

**C**

**3 D 1 E**

**Cluster Evolving Values**

C(A) {A} **1**

**B**

**3**

**2 1**

C(B) C(C) {B} {C} **G**

**F**

C(D) {D} C(E) {E} Step 1:

C(F) {F} Sort the edges and initialize the

C(G) {G} clusters

Weighted Graphs 43

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, ...}

**1**

**A2**

**C**

**3 D 1 E**

**Cluster Evolving Values**

C(A) {A, B} **1**

**B**

**3**

**2 1**

C(B) C(C) {A, B}

{C} **G**

**F**

C(D) {D} C(E) {E} Step 2:

C(F) {F} C(A) ≠ C(B)

C(G) {G} add AB to T, merge C(A) and C(B)

Weighted Graphs 44

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, ...} **1 A2**

**C**

**3 D 1 E**

**Cluster Evolving Values**

C(A) {A, B, C} **1**

**B**

**3**

**2 1**

C(B) C(C) {A, B, C} {A, B, C} **G**

**F**

C(D) {D} C(E) {E} Step 3:

C(F) {F} C(B) ≠ C(C)

C(G) {G} add BC to T, merge C(B) and C(C)

Weighted Graphs 45

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, DE, ...}

**1**

**A2**

**C**

**3 D 1 E**

**Cluster Evolving Values**

C(A) {A, B, C} **1**

**B**

**3**

**2 1**

C(B) C(C) {A, B, C} {A, B, C} **G**

**F**

C(D) {D, E} C(E) {D, E} Step 4:

C(F) {F} C(D) ≠ C(E)

C(G) {G} add DE to T, merge C(D) and C(E)

Weighted Graphs 46

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, DE, EF, ...}

**1**

**A2**

**C**

**3 D 1 E**

**Cluster Evolving Values**

C(A) {A, B, C} **1**

**B**

**3**

**2 1**

C(B) C(C) {A, B, C} {A, B, C} **G**

**F**

C(D) {D, E, F} C(E) {D, E, F} Step 5:

C(F) {D, E, F} C(E) ≠ C(F)

C(G) {G} add EF to T, merge C(E) and C(F)

Weighted Graphs 47

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, DE, EF, ...}

**1**

**A2**

**C**

**3 D 1 E**

**Cluster Evolving Values**

C(A) {A, B, C} **1**

**B**

**3**

**2 1**

C(B) C(C) {A, B, C} {A, B, C} **G**

**F**

C(D) {D, E, F} C(E) {D, E, F} Step 6:

C(F) {D, E, F} C(A) = C(C) , discard AC

C(G) {G}

Weighted Graphs 48

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, DE, EF, ...}

**1**

**A2**

**C**

**3 D 1 E**

**Cluster Evolving Values**

C(A) {A, B, C} **1**

**B**

**3**

**2 1**

C(B) C(C) {A, B, C} {A, B, C} **G**

**F**

C(D) {D, E, F} C(E) {D, E, F} Step 7:

C(F) {D, E, F} C(D) = C(F) , discard DF

C(G) {G}

Weighted Graphs 49

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, DE, EF, CD, ...}

**1**

**A2**

**C**

**3 D 1 E**

**Cluster Evolving Values** C(A) {A, B, C, D, E, F} **1**

**B**

**3**

**2 1**

C(B) {A, B, C, D, E, F} C(C) {A, B, C, D, E, F} **G**

**F**

C(D) {A, B, C, D, E, F} C(E) {A, B, C, D, E, F} Step 7:

C(F) {A, B, C, D, E, F} C(C) ≠ C(D)

C(G) {G} add CD to T, merge C(C) and C(D)

Weighted Graphs 50

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, DE, EF, CD, DG, ...}

**1**

**A2**

**C**

**3 D 1 E**

**Cluster Evolving Values**

C(A) {A, B, C, D, E, F, G} **1**

**B**

**3**

**2 1**

C(B) {A, B, C, D, E, F, G} C(C) {A, B, C, D, E, F, G} **G**

**F**

C(D) {A, B, C, D, E, F, G} C(E) {A, B, C, D, E, F, G} Step 8:

C(F) {A, B, C, D, E, F, G} C(D) ≠ C(G)

C(G) {A, B, C, D, E, F, G} add DG to T, merge C(D) and C(G)

Weighted Graphs 51

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, DE, EF, CD, DG}

**1**

**A2**

**C**

**3 D 1 E**

**Cluster Evolving Values**

C(A) {A, B, C, D, E, F, G} **1**

**B**

**3**

**2 1**

C(B) {A, B, C, D, E, F, G} C(C) {A, B, C, D, E, F, G} **G**

**F**

C(D) {A, B, C, D, E, F, G} C(E) {A, B, C, D, E, F, G} Now we have n-1 = 6 edges in T, the

C(F) {A, B, C, D, E, F, G} algorithm stops.

C(G) {A, B, C, D, E, F, G}

Weighted Graphs 52

Correctness: Background Facts

Suppose G = (V, E) is a connected simple graph.

A. Suppose V = some V1 i U ≠ VVj, 12 , x U V∈ 2. , . V. . i . and U ., VVkk y . are Then ∈ disjoint subsets of V with k > 1, and that there is an edge (x,y) in E such that for Vj. (From the labs)

B. Suppose vertices in S common = (VS, ES) (in and other any edge (x,y) in E for which U VT, ES U ET U {(x,y)}) is also T = words, x (Va ∈ tree. TV, S Eand VT) S (From are and y subtrees of G ∈ VVT the Tare , the labs)

disjoint). subgraph with no

Then for U = (VS C. Suppose W is a subset of the set E of edges of G and |W| < n - 1. Consider the subgraph H of G formed from W by defining the edges of H to be W and defining the vertices of H to be the endpoints of those edges, and assume that H contains no cycle. Then there exists an edge (x,y) in G not in W so that the graph formed by W U {(x,y)} also contains no cycle.

53

Proof of Background Fact C

Shortest Paths 54

Correctness

We show that Kruskal produces a spanning tree (proof that it is a minimal

spanning tree is optional).

1. During execution, for each cluster C, the edges in T that have endpoints in C form a spanning tree in G[C]. Moreover, during execution, if two endpoints in C were joined by a new edge, it would create a cycle (therefore, this step never allowed by the algorithm)

Proof: This is true when the first edge is examined (one edge, two vertices). Assuming true after the first i stages of the algorithm, at the next stage if edge uv is being examined and C(u) not equal to C(v), then the an edge is added with one vertex in the tree in C(u) and one in the tree in C(v). By Fact B, a new tree is formed that is a spanning tree for C(u) U C(v). For the moreover clause, if two endpoints in a cluster were joined by a new edge, two endpoints of its spanning tree would be joined and the resulting subgraph would contain a cycle.

Shortest Paths 55

Correctness

2. The main loop terminates (it is conceivable that after all edges have been examined, T still contains < n – 1 edges – this is shown to be impossible).

Proof: Suppose T still contains < n – 1 edges after processing every edge. By Fact C, there is an edge e that can be added to T without creating a cycle. But the algorithm visited every edge (including e) and it rejected e. The algorithm rejects an edge only because it would create a cycle, but e does not create a cycle -- contradiction.

Shortest Paths 56

Correctness

3. At the end of the algorithm, T is a spanning tree.

Proof. We have shown that T consists of only disjoint trees, so T contains no cycle, and T has n-1 edges. We first show that T has n vertices.

Let

mT = # edges in T and nT = # vertices in T. Then mT = n-1. If n > nT, it follows that mT = n - 1 ≥ nT and so T must contain a cycle (which is impossible). Therefore, T has n vertices and so is a spanning subgraph. Since mT = nT -1 and T has no cycles, T must be a tree and is therefore a spanning tree.Shortest Paths 57

OPTIONAL: Proof that Output is a Minimal Spanning Tree

We establish the following loop invariant I(i):

At each stage i of the algorithm, if T is the collection of edges obtained so far, there is a minimum spanning tree for G that contains T

Assuming we can establish this loop invariant, then, when the algorithm finishes, it will follow from the loop invariant that there is an MST that contains T. But, as shown in previous slides, when the while loop of the algorithm ends, T has become a spanning tree. It follows that the MST that contains T must be T itself.

Shortest Paths 58

Proof of Loop Invariant

The invariant holds at the start – G must have an MST since, as we have shown, it does have a spanning tree, so one such spanning tree must have minimal weight.

Assuming the invariant holds so far, we consider the next step of the algorithm in which the edge (x,y) is considered; assume that C(x) ≠ C(y), so that the algorithm will add (x,y) to T, the set of edges obtained so far. By induction hypothesis, there is an MST for G that contains T; let us denote this MST Tmst. We show there is an MST for G that contains T U {(x,y)}.

Shortest Paths 59

Proof (continued)

Case I. The new edge (x,y) happens to belong to Tmst. In that case, Tmst also contains T U {(x,y)}, and the induction step is complete.

Case II. The new edge (x,y) does not belong to Tmst. Since Tmst is a spanning tree, both x and y are vertices in Tmst. Therefore Tmst U {(x,y)} contains n edges, and so contains a cycle C, with (x,y) as one of its edges (if (x,y) were not one of the edges of C, then C would be a subgraph of Tmst). Recall T U {(x,y)} contains no cycle (by construction). So there must be some (u,v) in C (and also in Tmst) that does not belong to T (since edges of C are not a subset of the edges of T). Define T' by

T' = Tmst U {(x,y)} – {(u,v)}

60

Proof (continued)

**Claim 1.** T' is a spanning tree.

**Proof.** Suppose r, s are vertices in G. Since T is a spanning tree, there is a path p from r to s. If this path begins like this: r, . . ., v, u, . . ., then we can replace the occurrence of the edge v,u in p with a path in C from v back to u, and then continue to follow p. Likewise if it begins r, . . ., u, v, ... Therefore, T' is connected and clearly has n – 1 edges (since T has that many); it follows that T' is a tree that visits every vertex.

61

Proof (continued)

**Claim 2**. T' = Tmst U {(x,y)} – {(u,v)} has the same weight as Tmst.

**Proof**. Note that T U {(u,v)} contains no cycle, since Tmst includes T U{(u,v)}. So the algorithm could not have already rejected (u,v) [it would reject (u,v) only if (u,v) would introduce a cycle into T]. Since edges are ordered by weight, this means that (u,v) must have weight greater than or equal to wt(x,y). It follows that wt(T') ≤ wt(Tmst). But Tmst is an MST, so we also have wt(Tmst) ≤ wt(T'). This proves Claim 2.

**Continuation of Proof of Fact 5**. Claims 1 and 2 show that T' is also an MST, but now T' includes both T and (x,y). This establishes the induction step of the proof of the loop invariant I. As observed earlier, we may now conclude that Fact 5 holds and that the algorithm produces an MST.

Shortest Paths 62

Running Time of Kruskal

Computation

∎ time to sort edges: O(mlog n) ∎ time for while loop O(mn)

◆ for each edge (x,y):

- comparison C(x) = C(y), with a hashtable implementation of sets, is O(n) - merging C(x), C(y) costs min{|C(x)|, |C(y)|}, which is O(n). Cost of while loop can be improved with a good choice of

data structure

Shortest Paths 63

DisjointSets Data Structure

Data

) for maintaining a partition of a set into disjoint subsets (this data structure is sometimes called Partition rather than DisjointSets) General features

∎ Data:

◆ Universe U – the base set that is being partitioned (this set is

never altered) ◆ Collection

shrinks because of repeated union operations) ∎ Operations:

◆ find(x) – returns the subset Xi to which x belongs ◆ union(A,B) – replaces the subsets A, B in

structure (U,

= {X1, X2, ..., Xn} of subsets of the universe – the subsets are disjoint and their union is U (these subsets are modified when the data structure is used – size of

with A U B.

64

Example

find Operation:

find(2) = X1 find(5) = X3

union Operation:

union(X1, X2) = X1 U X2 = {1,2,3} new value for

Initial Structure: ∎ U = {1, 2, 3, 4, 5} ∎ X1 = {1, 2}, X2 = {3}, X3 = {4, 5}

∎

= {X1, X2, X3}

is {{1,2,3}, {4,5}}

Shortest Paths 65

Tree-Based Implementation of DisjointSets

The elements of each set X in the collection

are represented by nodes in a tree TX; the set X itself is referenced by its root rX. find(x) returns the root of the tree to which x belongs

x,y) union(

joins the tree that x belongs to to the tree that y belongs to by pointing root of one to the root of the other.

Shortest Paths 66

Example

U = {'a', 'b', 'c', 'd', 'e'}

= {{'a'}, {'b'}, {'c', 'd', 'e'}} Tree representations:

• find('d') returns 'c'

Shortest Paths 67

Example (cont)

• union('b', 'c') points root 'b' to 'c'

Now, find('b') returns 'c'

68

Code

//handle trees by keeping track of parents only //whenever a character c is a root, its parent is set to be c itself HashMap<Character, Character> parents = new HashMap<Character, Character>(); char[] universe;

//find returns the root of tree representing a subset, to which element belongs //worst case: find requires full depth of tree to locate root of representing tree public char find(char element) {

char nextParent = parents.get(element); if(nextParent == element) {

return element; } else {

return find(nextParent); } }

Shortest Paths 69

Code

//union() accepts only tree roots (representing subsets) as arguments //The method simply points the first root to the second //In the worst case, resulting tree is taller than original two public void union(char a\_tree, char b\_tree) {

parents.put(a\_tree, b\_tree); }To avoid building up trees that are too tall (and therefore imbalanced), an optimization can be used: Always point the shorter tree’s root to that of the taller.

bbaU dadc

e f

c

e f

Shortest Paths 70

Optimized Code

HashMap<Character, Character> parents = new HashMap<Character, Character>(); char[] universe; //keep track of heights of trees HashMap<Character, Integer> heights = new HashMap<Character, Integer>();

public void union(char a\_tree, char b\_tree) {

int height\_a = heights.get(a\_tree); int height\_b = heights.get(b\_tree); if(height\_a < height\_b) {

parents.put(a\_tree, b\_tree); } else if(height\_b < height\_a) {

parents.put(b\_tree, a\_tree); } else { //height\_a == height\_b

parents.put(a\_tree, b\_tree); heights.put(b\_tree, height\_b + 1); //this is case in which height is increased } }NOTE: With this optimization of union(), find() can be shown to run in O(log n) in the worst case. See https://en.wikipedia.org/wiki/Disjoint-set\_data\_structure

71

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {...}

**A**Step 1: **1**

Sort the edges and initialize the clusters

**B**

**2**

**C**

**3 D 1 E**

**1**

**3**

**2 1**

**G**

**F**

72

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, ...}

**A1**

Step 2: C(A) ≠ C(B) add AB to T, merge C(A) and C(B)

**2**

**2**

**C**

**C**

**3 D 1 E**

**3 D 1 E**

**3 D 1 E**

**3 D 1 E**

**1**

**2 1**

**2 1**

**B**

**3**

**2 1**

**G**

**F**

73

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, ...} **A1**

Step 3: C(B) ≠ C(C) add BC to T, merge C(B) and C(C)

**B**

**2**

**C**

**3 D 1 E**

**1**

**3**

**2 1**

**G**

**F**

74

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, DE, ...}

**A**Step 4: **1**

C(D) ≠ C(E) add DE to T, merge C(D) and C(E)

**B**

**2**

**C**

**3 D 1 E**

**1**

**3**

**2 1**

**G**

**F**

75

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, DE, EF, ...}

**A**Step 5: **1**

C(E) ≠ C(F) add EF to T, merge C(E) and C(F)

**B**

**2**

**C**

**3 D 1 E**

**1**

**3**

**2 1**

**G**

**F**

[point root of F to root of E]

76

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, DE, EF, ...}

**A1**

Step 6: C(A) = C(C) , discard AC

**2**

**2**

**C**

**C**

**3 D 1 E**

**3 D 1 E**

**3 D 1 E**

**3 D 1 E**

**B**

**3**

**2 1**

**G**

**F**

**1**

**2 1**

Weighted Graphs 77

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, DE, EF, ...}

**A**Step 7:

**1**

C(D) = C(F) , discard DF

**B**

**2**

**C**

**3 D 1 E**

**1**

**3**

**2 1**

**G**

**F**

Weighted Graphs 78

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, DE, EF, CD, ...}

**A**Step 7:

**1**

C(C) ≠ C(D) add CD to T, merge C(C) and C(D)

**B**

**2**

**C**

**3 D 1 E**

**1**

**3**

**2 1**

**G**

**F**

79

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG T = {AB, BC, DE, EF, CD, DG, ...}

**A**Step 8:

**1**

C(D) ≠ C(G) add DG to T, merge C(D) and C(G)

**B**

**2**

**C**

**3 D 1 E**

**1**

**3**

**2 1**

**G**

**F**

80