CSE 326 Data Structures

Dave Bacon

Graphs

Logisitics

- Turn in Homework 6…

- Project 3 code due on Monday!
- Project 3 writeup due on Thursday!

- Homework 7 will be due….

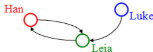
- Read Chapter 9 of Weiss

- Complain about the class on the survey on the webpage!

Graph… ADT?

- Not quite an ADT… operations not clear
- A formalism for representing relationships between objects

  Graph `G = (V,E)`
  - *Set of vertices*:
    $V = \{v_1, v_2, \ldots, v_n\}$
  - *Set of edges*:
    $E = \{e_1, e_2, \ldots, e_m\}$
    where each e_i connects two vertices (v_{i1}, v_{i2})

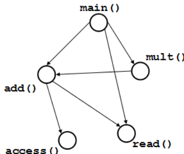



```
V = {Han, Leia, Luke}
E = {(Luke, Leia),
     (Han, Leia),
     (Leia, Han)}
```

Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no (directed) cycles.

Aside: *If program call-graph is a DAG, then all procedure calls can be in-lined*



Graph Representations

0. List of vertices + list of edges
1. 2-D matrix of vertices (marking edges in the cells)
   "adjacency matrix"
2. List of vertices each with a list of adjacent vertices
   "adjacency list"

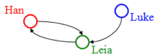
Vertices and edges may be labeled

Things we might want to do:

- iterate over vertices
- iterate over edges
- iterate over vertices adj. to a vertex
- check whether an edge exists

Representation 1: Adjacency Matrix

A `|V| x |V|` array in which an element `(u,v)` is true if and only if there is an edge from `u` to `v`



	Han	Luke	Leia
Han			
Luke			
Leia			

space requirements:

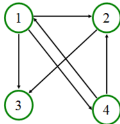
runtime:

Representation

- adjacency **matrix**:

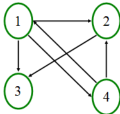
$$A[u][v] = \begin{cases} \text{weight} & \text{, if } (u, v) \in E \\ 0 & \text{, if } (u, v) \notin E \end{cases}$$

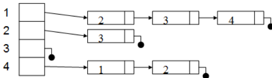
	1	2	3	4
1				
2				
3				
4				



Representation

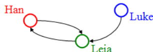
- adjacency **list:**

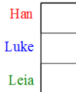




Representation 2: Adjacency List

A `|V|`-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices

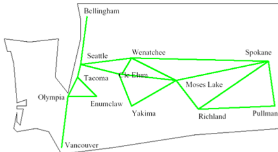




space requirements:

runtime:

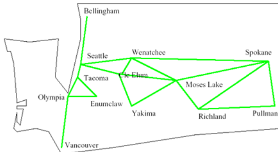
Some Applications: Moving Around Washington



What's the *shortest way* to get from Seattle to Pullman?

Edge labels:

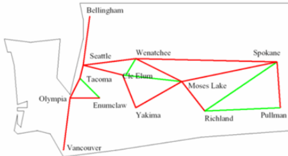
Some Applications: Moving Around Washington



What's the *fastest way* to get from Seattle to Pullman?

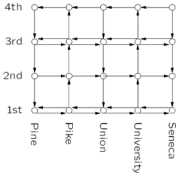
Edge labels:

Some Applications: Reliability of Communication



If Wenatchee's phone exchange *goes down*, can Seattle still talk to Pullman?

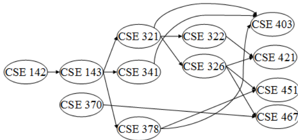
Some Applications:
Bus Routes in Downtown Seattle



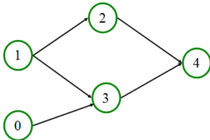
If we're at 3rd and Pine, how can we get to
1st and University using Metro?

Application: Topological Sort

Given a directed graph, `G = (V,E)`, output all the vertices in `V` such that no vertex is output before any other vertex with an edge to it.



Is the output unique?



Valid Topological Sorts:

Topological Sort: Take One

1. Label each vertex with its *in-degree* (# of inbound edges)
2. **While** there are vertices remaining:
   a. Choose a vertex *v* of *in-degree zero*; output *v*
3. Reduce the in-degree of all vertices adjacent to *v*
   a. Remove *v* from the list of vertices

Runtime:

```
void Graph::topsort(){
  Vertex v, w;

  labelEachVertexWithItsInDegree();

  for (int counter=0; counter < NUM_VERTICES;
                                   counter++){
    v = findNewVertexOfDegreeZero();

    v.topologicalNum = counter;
    for each w adjacent to v
      w.indegree--;
  }
}
```

Topological Sort: Take Two

1. Label each vertex with its in-degree
2. Initialize a queue *Q* to contain all in-degree zero vertices
3. While *Q* not empty
   a. *v* = *Q*.dequeue; output *v*
   b. Reduce the in-degree of all vertices adjacent to *v*
   c. If new in-degree of any such vertex *u* is zero *Q*.enqueue(*u*)

<u>Note</u>: could use a stack, list, set, box, … instead of a queue

Runtime:

```
void Graph::topsort(){
  Queue q(NUM_VERTICES);  int counter = 0; Vertex v, w;
  labelEachVertexWithItsIn-degree();

  q.makeEmpty();
  for each vertex v
    if (v.indegree == 0)
      q.enqueue(v);

  while (!q.isEmpty()){
    v = q.dequeue();
    v.topologicalNum = ++counter;
    for each w adjacent to v
      if (--w.indegree == 0)
        q.enqueue(w);
  }
}
```

intialize the queue

get a vertex with indegree 0

insert new eligible vertices

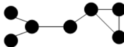
Runtime:

Graph Traversals

- Breadth-first search (and depth-first search) work for arbitrary (directed or undirected) graphs - not just mazes!
  - Must mark visited vertices so you do not go into an infinite loop!
- Either can be used to determine connectivity:
  - Is there a path between two given vertices?
  - Is the graph (weakly) connected?
- Which one:
  - Uses a queue?
  - Uses a stack?
  - Always finds the shortest path (for unweighted graphs)?

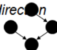
Graph Connectivity

Undirected graphs are *connected* if there is a path between any two vertices



Directed graphs are *strongly connected* if there is a path from any one vertex to any other

Directed graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*



A *complete* graph has an edge between every pair of vertices

The Shortest Path Problem

Given a graph G, edge costs $c_{i,j}$, and vertices s and t in G, find the shortest path from s to t.

For a path $p = v_0\ v_1\ v_2 \ldots v_k$

- *unweighted length* of path $p = k$ (a.k.a. *length*)
- *weighted length* of path $p = \sum_{i=0..k-1} c_{i,i+1}$ (a.k.a *cost*)

Path length equals path cost when ?

Single Source Shortest Paths (SSSP)

Given a graph *G*, edge costs $c_{i,j}$, and vertex *s*, find the shortest paths from *s* to all vertices in G.

- Is this harder or easier than the previous problem?

All Pairs Shortest Paths (APSP)

Given a graph *G* and edge costs $c_{i,j}$, find the shortest paths between all pairs of vertices in G.

- Is this harder or easier than SSSP?

- Could we use SSSP as a subroutine to solve this?

Variations of SSSP

- Weighted vs. unweighted
- Directed vs undirected
- Cyclic vs. acyclic
- Positive weights only vs. negative weights allowed
- Shortest path vs. longest path
- …

Applications

- Network routing
- Driving directions
- Cheap flight tickets
- Critical paths in project management (see textbook)
- …

SSSP: Unweighted Version

Ideas?

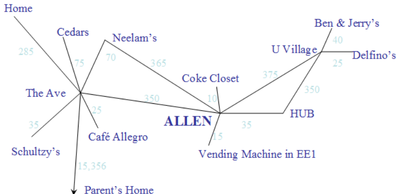
```
void Graph::unweighted (Vertex s){
  Queue q(NUM_VERTICES);
  Vertex v, w;
  q.enqueue(s);
  s.dist = 0;

  while (!q.isEmpty()){
    v = q.dequeue();
    for each w adjacent to v
      if (w.dist == INFINITY){
        w.dist = v.dist + 1;
        w.path = v;
        q.enqueue(w);
      }
  }
}
```

each edge examined at most once – if adjacency lists are used

each vertex enqueued at most once

total running time: O(   )

Weighted SSSP: The Quest For Food



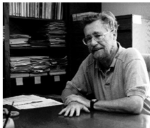
Can we calculate shortest distance to all nodes from Allen Center?

Dijkstra, Edsger Wybe

Legendary figure in computer science; was a professor at University of Texas.

Supported teaching introductory computer courses without computers (pencil and paper programming)

Supposedly wouldn't (until very late in life) read his e-mail; so, his staff had to print out messages and put them in his box.



E.W. Dijkstra (1930-2002)

1972 Turning Award Winner,
Programming Languages, semaphores, and …

CSE 326 Data Structures

Dave Bacon

Shortest Path and Djistra's Algorithm

Logisitics

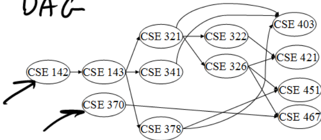
- Project 3 code due tonight!
- Project 3 writeup due on Thursday!

- Homework 7 will be due Wed, March 7

- Read Chapter 9 of Weiss

Application: Topological Sort

Given a directed graph, `G = (V,E)`, output all the vertices in `V` such that no vertex is output before any other vertex with an edge to it.



Is the output unique? NO

Topological Sort: Take Two

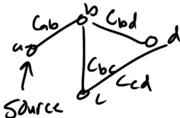
1. Label each vertex with its in-degree
2. Initialize a queue *Q* to contain all in-degree zero vertices
3. While *Q* not empty
   a. *v* = *Q*.dequeue; output *v*
   b. Reduce the in-degree of all vertices adjacent to *v*
   c. If new in-degree of any such vertex *u* is zero *Q*.enqueue(*u*)

<u>Note</u>: could use a stack, list, set, box, … instead of a queue

Runtime: O(|V|+|E|)

Single Source Shortest Paths (SSSP)

Given a graph *G*, edge costs $c_{i,j}$, and vertex *s*, find the shortest paths from *s* to all vertices in G.



SSSP: Unweighted Version

Ideas?

Given: G

Source: s.

$c_{ij} = 1$ iff $(i, j) \in E$

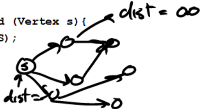
$= 0$ otherwise

Use BFS

B
r
e
a
d
t
h

F
i
r
s
t

```
void Graph::unweighted (Vertex s){
  Queue q(NUM_VERTICES);
  Vertex v, w;
  q.enqueue(s);
  s.dist = 0;

  while (!q.isEmpty()){
    v = q.dequeue();
    for each w adjacent to v
      if (w.dist == INFINITY){
        w.dist = v.dist + 1;
        w.path = v;
        q.enqueue(w);
      }
  }
}
```



each edge examined at most once – if adjacency lists are used

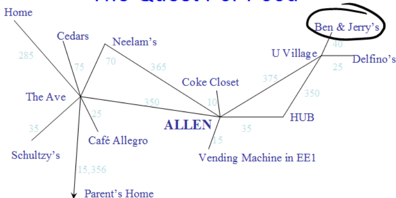
each vertex enqueued at most once

path

|V|+|E|

total running time: O(   )

Weighted SSSP: The Quest For Food



Can we calculate shortest distance to all nodes from Allen Center?

Dijkstra, Edsger Wybe

Legendary figure in computer science; was a professor at University of Texas.

Supported teaching introductory computer courses without computers (pencil and paper programming)

Supposedly wouldn't (until very late in life) read his e-mail; so, his staff had to print out messages and put them in his box.

E.W. Dijkstra (1930-2002)

"Go To Statement Considered Harmful",

http://www.cs.utexas.edu/users/EWD/

1972 Turning Award Winner, Programming Languages, semaphores, and …

Dijkstra, Edsger Wybe

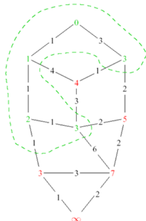
Three Golden Rules for Successful Scientific Research

1st Rule:
"Raise your quality standards as high as you can live with, avoid wasting your time on routine problems, and always try to work as closely as possible at the boundary of your abilities. Do this, because it is the only way of discovering how that boundary should be moved forward."

2nd Rule:
"We all like our work to be socially relevant and scientifically sound. If we can find a topic satisfying both desires, we are lucky; if the two targets are in conflict with each other, let the requirement of scientific soundness prevail."

3rd Rule:
"Never tackle a problem of which you can be pretty sure that (now or in the near future) it will be tackled by others who are, in relation to that problem, at least as competent and well-equipped as you."

Dijkstra's Algorithm: Idea

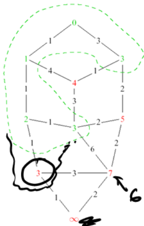


Adapt BFS to handle weighted graphs

Two kinds of vertices:

- Finished or known vertices
  - Shortest distance has been computed
- Unknown vertices
  - Have tentative distance

Dijkstra's Algorithm: Idea



At each step:

1) Pick closest unknown vertex
2) Add it to known vertices
3) Update distances

Dijkstra's Algorithm: Pseudocode

Initialize the cost of each node to ∞

Initialize the cost of the source to 0

While there are unknown nodes left in the graph
- Select an unknown node *b* with the lowest cost
- Mark *b* as known
- For each node *a* adjacent to *b*
  - *a*'s cost = min(*a*'s old cost, *b*'s cost + cost of (*b*, *a*))

priority queue

```
void Graph::dijkstra(Vertex s){
  Vertex v,w;

  Initialize s.dist = 0 and set dist of all other
  vertices to infinity

  while (there exist unknown vertices, find the
  one b with the smallest distance)
    b.known = true;

    for each a adjacent to b
      if (!a.known)
        if (b.dist + Cost_ba < a.dist){
          decrease(a.dist to= b.dist + Cost_ba);
          a.path = b;
        }
  }
}
```

Student Activity

s V_0 0, V_1 6, V_2 2, V_3 1, V_4 2, V_5, V_6 6

2, 1, 5, 1, 1, 1, 2, 6, 5, 3, 10

V	Kwn	Dist	path
~~v0~~	~~F~~	~~0~~	~~–~~
v1	F	∞ 6	–
v2	~~F~~ T	~~∞~~ 2	–
~~v3~~	~~F T~~	~~∞ 1~~	~~–~~
v4	F	∞ 2	–
v5	F	∞ ~~7~~ 4	–
v6	F	∞ 6	–

Order declared Known:

0 Known

3 Known

Dijkstra's Alg: Implementation Unoptimizied

Array

Initialize the cost of each node to ∞

Initialize the cost of the source to 0

While there are unknown nodes left in the graph

- Select the unknown node *b* with the lowest cost ← scan V (costs)
- Mark *b* as known
- For each node *a* adjacent to *b*
  - *a*'s cost = min(*a*'s old cost, *b*'s cost + cost of (*b*, *a*))

V times

E * O(1)

Running time? $O(|V|^2 + |E|)$

Dijkstra's Alg: Implementation Optimized

Initialize the cost of each node to ∞

Initialize the cost of the source to 0

While there are unknown nodes left in the graph ← V times

- Select the unknown node *b* with the lowest cost ← heap — deleteMin $\log|V|$
- Mark *b* as known
- For each node *a* adjacent to *b*
  - *a*'s cost = min(*a*'s old cost, *b*'s cost + cost of (*b*, *a*))

{ decrease key, hash table } $|E| \log|V|$

Running time?

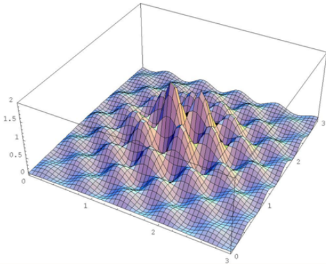
$O(|V| \log|V| + |E| \log|V|)$

Dijkstra's Algorithm: a Greedy Algorithm

Greedy algorithms always make choices that *currently* seem the best

- Short-sighted – no consideration of long-term or global issues
- Locally optimal - does not always mean globally optimal!!

Global and Local Optimum



Greedy

0.25 0.10 0.05 0.01

0.82

3 * 0.25

0.07 left

1 * 0.05

2 * 0.01

min #coins

0.12

0.15

1 * 0.12

0.03

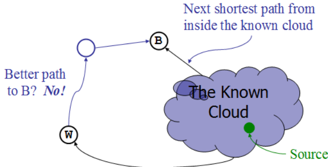
3 * 0.03

Correctness of Dijkstra's

Intuition for correctness:

- shortest path from source vertex to itself is 0
- cost of going to adjacent nodes is at most edge weights
- cheapest of these must be shortest path to that node
- update paths for new node and continue picking cheapest path

Correctness: The Cloud Proof



How does Dijkstra's decide which vertex to add to the Known set next???

- If path to **B** is shortest, path to **W** must be *at least as long* *(or else we would have picked* **W** *as the next vertex)*
- So any path *through* **W** to **B** *cannot* be any shorter!

Correctness: Inside the Cloud

Prove by induction on # of nodes in the cloud:

- Initial cloud is just the source with shortest path 0
- Assume: Everything inside the cloud has the correct shortest path
- Inductive step: Only when we prove the shortest path to some node ***v*** (which is *not* in the cloud) is correct, we add it to the cloud

When does Dijkstra's algorithm not work?

Dijkstra's vs BFS

At each step:

1) Pick closest unknown vertex
2) Add it to finished vertices
3) Update distances

Dijkstra's Algorithm

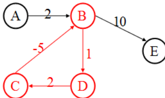
At each step:

1) Pick vertex from queue
2) Add it to visited vertices
3) Update queue with neighbors

Breadth-first Search

Some Similarities:

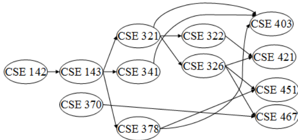
The Trouble with Negative Weight Cycles



What's the shortest path from A to E?

Problem?

Acyclic Graphs?



Minimum Spanning Trees

Given an undirected graph **G**=(**V**,**E**), find a graph **G'=(V, E')** such that:

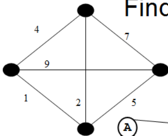
- E' is a subset of E
- |E'| = |V| - 1
- G' is connected
- $\sum_{(u,v)\in E'} c_{uv}$ is minimal

G' is a **minimum spanning tree**.

Applications: wiring a house, power grids, Internet connections

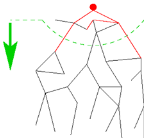
Student Activity

Find the MST



A
1
B
7
F
6
H
3
5
2
12
11
9
C
G
4
10
13
D
4
E

Two Different Approaches



Prim's Algorithm

Almost identical to Dijkstra's

Kruskals's Algorithm

Completely different!