Priority Queue Implementations and Minimum Spanning Trees

CSE 101: Design and Analysis of Algorithms
Lecture 6
CSE 101: Design and analysis of algorithms

• Priority queue implementations and minimum spanning trees
  – Reading: Sections 4.5 and 5.1

• Homework 2 is due today, 11:59 PM

• Quiz 1 is Oct 18, in class
Dijkstra’s algorithm, towards low-level

procedure dijkstra(G, ℓ, s)
X: set of vertices we know distances to
F: set of vertices we aren’t sure of the distance to

• Let X and F be sets
• All vertices start out in F with dist set to ∞
• Set dist(s)=0 and move s from F to X
• Repeat until F is empty or there are no edges from X to F:
  – Let w be the vertex in F with the minimum value:
    \[ \text{dist}(v) + ℓ(v, w) \]
    for all v in X
  – Set \[ \text{dist}(w) = \text{dist}(v) + ℓ(v, w) \]
    (Note: only need to keep track of best current edge to each vertex in F, so only one number per node in F. Can use same array, dist to keep track of this number)
  – Move w from F to X

Based on slides courtesy of Miles Jones
Dijkstra’s algorithm, towards low-level

procedure dijkstra(G, ℓ, s)
X: set of vertices we know distances to
F: set of vertices we aren’t sure of the distance to

• Let X and F be sets
• All vertices start out in F with dist set to ∞
• Set dist(s)=0 and move s from F to X
  We will maintain the invariant that for all u ∈ F, dist(u) is the minimum over all e = (v,u), v ∈ X of dist(v)+ℓ(e)
• Repeat until F is empty or there are no edges from X to F:
  – Let w be the vertex in F with the minimum value:
    \[ dist(v) + ℓ(v,w) \]
    for all v in X
  – Set dist(w) = dist(v) + ℓ(v,w)
  – Move w from F to X
Dijkstra’s algorithm, towards low-level

procedure dijkstra(G, ℓ, s)
X: set of vertices we know distances to
F: set of vertices we aren’t sure of the distance to

• Let X and F be sets
• All vertices start out in F with dist set to ∞
• Set dist(s)=0 and move s from F to X
  For each u in F, maintain array dist:
  \[ dist(u) = \min_{(v,u) \in E} \min_{v \in X} dist(v) + \ell(v, u) \]

• Repeat until F is empty or there are no edges from X to F:
  – Let w be the vertex in F with the minimum value:
    \[ dist(v) + \ell(v, w) \] for all v in X
  – Set \( dist(w) = dist(v) + \ell(v, w) \)
  – Move w from F to X
  – Update dist(u) for all neighbors of w
Structures in Dijkstra’s algorithm

• Graph G: How does it change? Where do we access it?
• Set X: How does it change? Where do we access it?
• Set F: How does it change? Where do we access it?
Structures in Dijkstra’s algorithm

- Graph G: no changes, need to list members
  - Adjacency list
- Set X: insert, check membership
  - Array of booleans
- Set F: Find and delete the element with minimum key, $\text{dist}(u)$. Decrease the keys of some elements $u'$. 
What do we keep track of?

• Instead of picking a theoretical minimum:
  – Let \( w \) be the vertex in \( F \) with the minimum value
    \[ \text{dist}(v) + \ell(v, w) \]
    for all \( v \) in \( X \)

• We only have to choose the vertex \( w \) that has the minimum dist value
Priority queue

• A priority queue is a data structure of a set of objects (vertices) along with key values for each object that can be changed (alarm settings). Additionally, it can support the following operations.
  – insert
  – deletemin
  – decreasekey
Array as a priority queue

- Array: indexed by vertex, giving key value
  \[\text{key}(A), \text{key}(B), \text{key}(C), \text{key}(D), \text{key}(E), \text{key}(F)\]

- insert
- deletemin
- decreasekey
Array as a priority queue

- Array: indexed by vertex, giving key value
  \( R = [\text{key(A)}, \text{key(B)}, \text{key(C)}, \text{key(D)}, \text{key(E)}, \text{key(F)}] \)

- Insert \( O(1) \)
  - \( \text{insert(v, value)}. \ R[v] = \text{value}. \)

- delete\text{em}in \( O(n) \)
  - Find minimum and change value to null

- decrease\text{e}\text{key} \( O(1) \)
  - \( \text{decrease\text{e}\text{key}(v, value)}. \ R[v] = \text{value}. \)

Let \( n = |V| \)
Initialize the array as an array of nulls
Array as a priority queue

• Dijkstra’s algorithm
  makequeue + deletemin × |V| + decreasekey × |E|
  – If we use an array, then it will take
    \[ O(|V|) + O(|V|) \cdot |V| + O(1) \cdot |E| = O(|V|^2 + |E|) = O(|V|^2) \]
Runtime of Dijkstra’s algorithm

procedure dijkstra(G,ℓ,s)
for all u in V
    dist(u) := infinity
    prev(u) := nil
dist(s) := 0
H := makequeue(V)
while H is not empty
    u := deletemin(H)
for all edges (u,v) in E
    if dist(v) > dist(u) + ℓ(u,v) then
        dist(v) := dist(u) + ℓ(u,v)
        prev(v) := u
        decreasekey(H,v)
Runtime of Dijkstra’s algorithm, array as priority queue

procedure dijkstra(G, ℓ, s)
for all u in V
  dist(u) := infinity
  prev(u) := nil
  dist(s) := 0
H := makequeue(V)
while H is not empty
  u := deletemin(H) \text{ executes } |V| \text{ times}
  for all edges (u,v) in E \text{ deg}(u)
    if dist(v) > dist(u) + ℓ(u,v) then
      dist(v) := dist(u) + ℓ(u,v)
      prev(v) := u
      decreasekey(H,v) \text{ executes } |E| \text{ times}

Initialize $O(|V|)$
deletemin \times |V| + \text{decreasekey} \times |E|$
= $O(|V|) \times |V| + O(1) \times |E|$
= $O(|V|^2)$
Binary heap

• A complete binary tree of objects (vertices) with the property that each key value of an object is less than the key value of its child
Binary heap

- The binary heap can be implemented with an array a[n] of vertices
- The children of $a_i$ are $a_{2i}$ and $a_{2i+1}$
- The parent of $a_i$ is $a_{i/2}$
Binary heap

- **deletemin**
  - The object with the minimum key value is guaranteed to be the root. Once you take it out, you must reorder the tree. You replace the root with the last object and let it trickle down.
Binary heap

- deletemin
  - The object with the minimum key value is guaranteed to be the root. Once you take it out, you must reorder the tree. You replace the root with the last object and let it trickle down.
Binary heap

- **deletemin**
  - The object with the minimum key value is guaranteed to be the root. Once you take it out, you must reorder the tree. You replace the root with the last object and let it trickle down.
Binary heap

- **deletemin**
  - The object with the minimum key value is guaranteed to be the root. Once you take it out, you must reorder the tree. You replace the root with the last object and let it trickle down.
Binary heap

• deletemin
  – When the last object is put in as the root, it may trickle down the entire length of the heap, the time taken is $O(\log(n))$ where $n$ is the number of objects in the heap
  – When performing Dijkstra’s algorithm, the number of objects in the heap is $|V|$ so the time taken is $O(\log(|V|))$
Binary heap

• decreasekey
  – When you decrease a key, you may have to adjust the heap by having the decreased key object bubble up
• **decreasekey**
  – When you decrease a key, you may have to adjust the heap by having the decreased key object bubble up
Binary heap

• decreasekey
  – When you decrease a key, you may have to adjust the heap by having the decreased key object bubble up
Binary heap

- decreasekey
  - When you decrease a key, you may have to adjust the heap by having the decreased key object bubble up.
Binary heap

• decreasekey
  – But, how do we know where v is in the binary heap?
  – Keep a supplemental array indexed by v, and keep pointers in both directions between this array and binary heap elements
Binary heap

• decreasekey
  – When the object decreases key, it may bubble up the entire heap, the time taken is $O(\log(n))$ where $n$ is the number of objects in the heap
  – When performing Dijkstra’s algorithm, the number of objects in the heap is $|V|$ so the time taken is $O(\log(|V|))$
Binary heap as a priority queue

• Dijkstra’s algorithm

\[ \text{makequeue} + \text{deletemin} \times |V| + \text{decreasekey} \times |E| \]

– If we use a binary heap, then it will take

\[ O(|V|) + O(\log(|V|)) |V| + O(\log(|V|)) |E| \]

\[ = O((|V| + |E|) \log(|V|)) \]
Dijkstra’s algorithm with different priority queues

- Runtime of array
  \( O(|V|^2) \)

- Runtime of binary heap
  \( O((|V| + |E|) \log(|V|)) \)

- Runtime of Fibonacci heap
  \( O(|V| \log(|V|) + |E|) \)
Advantages of flexibility

• By working with a higher level version of the algorithm, we can understand what is going on without getting caught up in details

• Flexibility allows us to fit data structures to be efficient for the given circumstance

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Binary heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense graphs: $</td>
<td>E</td>
<td>= O(</td>
</tr>
<tr>
<td>Sparse graphs</td>
<td>$O(</td>
<td>V</td>
</tr>
</tbody>
</table>
MINIMUM SPANNING TREES
Spanning trees

- A spanning tree of an undirected graph $G=(V,E)$ is a subgraph $G'=(V,E')$ such that $G'$ is a tree and all vertices in $V$ are connected.
- An output tree of depth-first search or breadth-first search is a spanning tree.
Example

• Suppose you have a network of computers that were linked pairwise
• Suppose each link has a positive maintenance cost
• Your job is to cut some links so that the cost of the network is minimized and the network stays connected
The minimized graph is a tree

- When is a connected undirected graph not a tree?
The minimized graph is a tree

• When is a connected undirected graph not a tree?
  – Suppose it was not a tree -> there is a cycle
  – Removing any cycle edge does not disconnect the graph
  – Removing any cycle edge will decrease cost without disconnection
Example

Greedy rule: which edge should we pick first?
Example

Greedy rule: which edge should we pick first?

delete the maximum edge if it doesn't disconnect the graph.
Greedy rule: which edge should we pick first?

keep adding in the smallest edge as long as it doesn't create a cycle until graph is connected.

connect two vertices already connected
Example

Greedy rule: which edge should we pick first?

*Pick the smallest weight edge*
Example

Greedy rule: which edge should we pick first?

*Pick the smallest weight edge*
Example

Greedy rule: which edge should we pick first?

*Pick the smallest weight edge unless its vertices are already connected*
Greedy rule: which edge should we pick first?

*Pick the smallest weight edge unless its vertices are already connected*
Greedy rule: which edge should we pick first?

*Pick the smallest weight edge unless its vertices are already connected*
Kruskal’s algorithm for finding the minimum spanning tree

• Start with a graph with only the vertices (no edges)
• Repeatedly add the next lightest edge that does not form a cycle
High-level to mid-level

• High level Kruskal’s algorithm
  – Given an undirected, connected graph with positive edge weights
  – Start with only the vertices
  – Repeat until graph is connected:
    • Add the lightest edge that does not create a cycle
      – Run depth-first search

• With your neighbors, discuss how to implement Kruskal’s algorithm
  – At this level, just describe what you want the data structures to do, not actually how to implement them. We will leave those details until later.
High-level to mid-level

- High level Kruskal’s algorithm
  - Given an undirected, connected graph with positive edge weights
  - Start with only the vertices
  - Repeat until graph is connected:
    - Add the lightest edge that does not create a cycle
      - Run depth-first search

- How do we know which edge is the lightest edge?

- How do we know that particular edge does not create a cycle?
High-level to mid-level

• High level Kruskal’s algorithm
  – Given an undirected, connected graph with positive edge weights
  – Start with only the vertices
  – Repeat until graph is connected:
    • Add the lightest edge that does not create a cycle
      – Run depth-first search

• How do we know which edge is the lightest edge?
  – Sort

• How do we know that particular edge does not create a cycle?
  – Remove $e$ from the graph to get a new graph $G’$. Run explore on $G’$ from one of the end-points of $e$ and if the other endpoint is visited, then return True, else return False.
How to implement Kruskal’s algorithm

Start with an empty graph $R$ (only vertices, no edges)
Sort edges by weight from smallest to largest $O(|E| \log |E|)$

For each edge $e$ in sorted order:
- If $e$ does not create a cycle in $R$ then $O(|V| + |E|)$, $|E|$ times
  - Add $e$ to $R$
- otherwise, do not add $e$ to $R$

Total time: $O(|E| \log |E|) + O(|E|(|V| + |E|))$

- How do we tell if adding an edge will create a cycle?
  - Remove $e$ from the graph to get a new graph $G'$. Run explore on $G'$ from one of the end-points of $e$ and if the other endpoint is visited, then return True, else return False. $O(|V| + |E|)$
Telling if an edge is in a cycle

• How do we tell if adding an edge will create a cycle?
• Homework 2, problem 2. $O(|V| + |E|)$
• Need to test for every edge, $|E|$ times
How to implement Kruskal’s algorithm

Start with an empty graph \( R \) (only vertices, no edges)
Sort edges by weight from smallest to largest \( O(|E| \log |E|) \)
For each edge \( e \) in sorted order:
  If \( e \) does not create a cycle in \( R \) then \( O(|V| + |E|) = O(|E|) \) see below
  Add \( e \) to \( R \)
otherwise
  do not add \( e \) to \( R \)

• Every graph we want to check is a forest so there are \( O(|V|) \) edges
• Since input is connected, \( |V| = O(|E|) \)
• In the worst case, we have to check every edge, so \( |E| \) times
Disjoint sets data structure (DSDS)

• What can it do?
• Given a set of objects, DSDS manage partitioning the set into disjoint subsets
• It does the following operations:
  – makeset(S): puts each element of S into a set by itself
  – find(u): returns the name of the subset containing u
  – union(u,v): unions the set containing u with the set containing v
Kruskal’s algorithm using a DSDS

procedure kruskal(G,w)
   Input: undirected connected graph G with edge weights w
   Output a set of edges X that defines a minimum spanning tree of G
   makeset(V)
   X = { }
   Sort the edges in E in increasing order by weight
   For all edges (u,v) in E
      if find(u) ≠ find(v):  
         Add edge (u,v) to X
         union(u,v)
         Separate connected components

Next lecture

• Minimum spanning trees and union-find
  – Reading: Section 5.1