Minimum Spanning Trees
Prim's Algorithm
Priority Queues

CSCI 3100

Review & Overview

Last time

- Spanning trees and spanning tree properties
- Minimum spanning tree of a Graph
- Kruskal's algorithm for MST uses greedy approach: select next smallest edge, if one of the end-points of that edge is not yet in the tree, add the edge and both end-points to the tree.

This time

- Prim's algorithm for MST
- Priority Queues
- Proof of Fluff

Prim's Algorithm

In Kruskal's algorithm we grow a *spanning forest* rather than a spanning tree

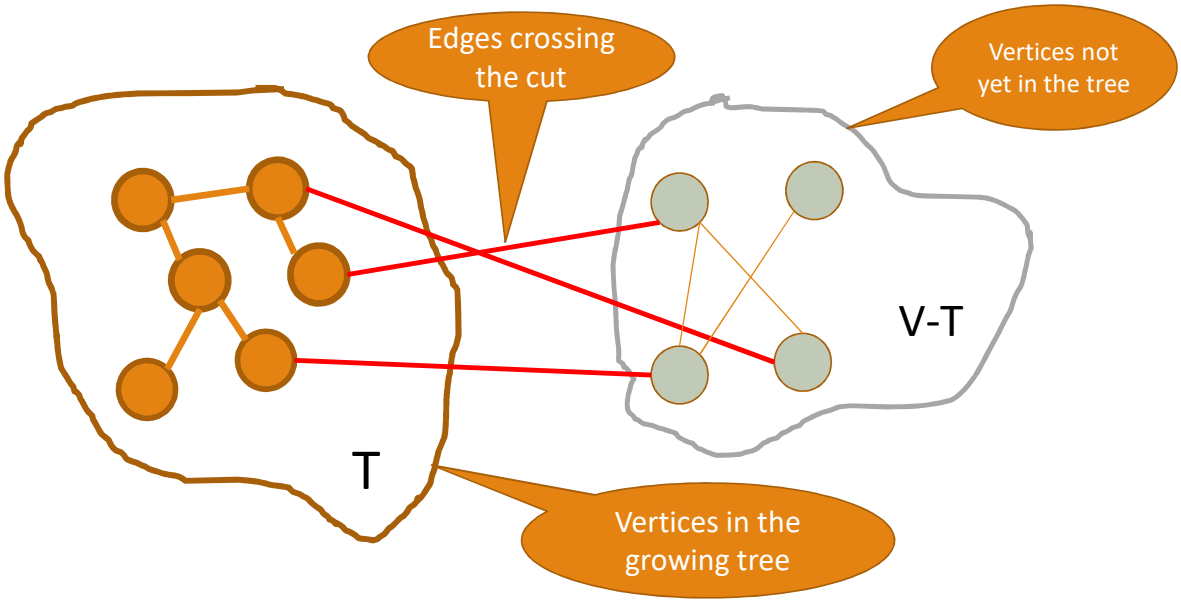
- Only at the end is it guaranteed to be connected, hence a *spanning tree*

In Prim's algorithm we grow a spanning tree

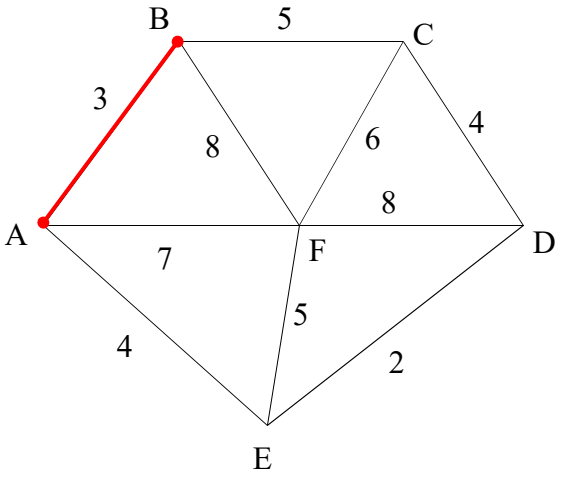
T = the set of nodes currently forming the tree

Prim's algorithm: always add the cheapest edge crossing the cut

3



Prim's Algorithm



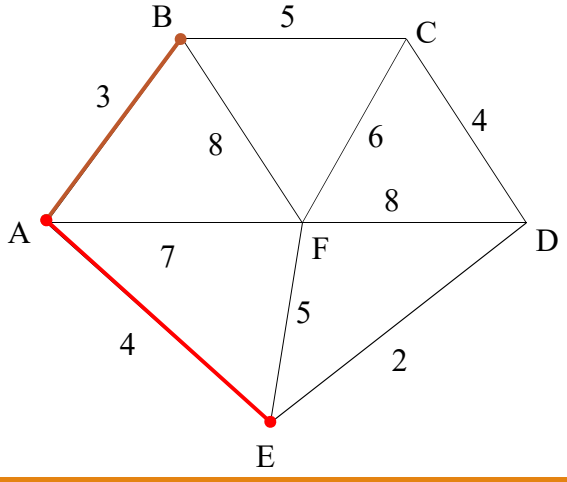
Select any vertex

A

Select the shortest edge connected to that vertex

AB 3

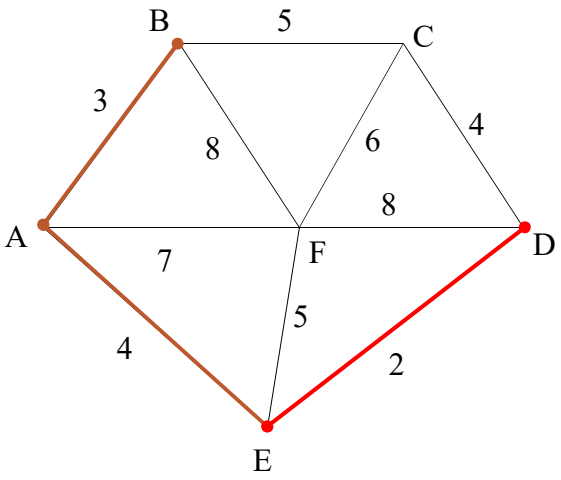
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

AE 4

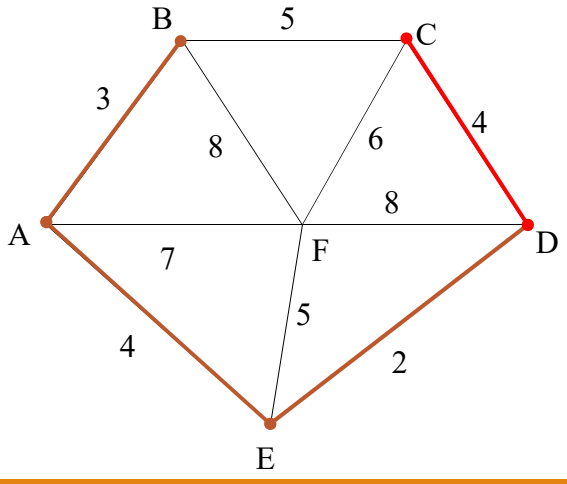
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

ED 2

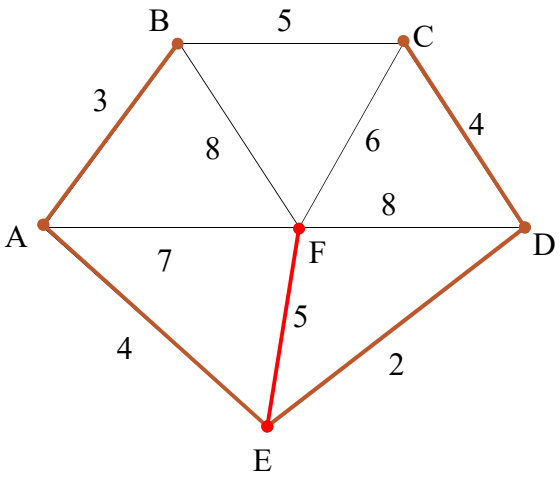
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

DC 4

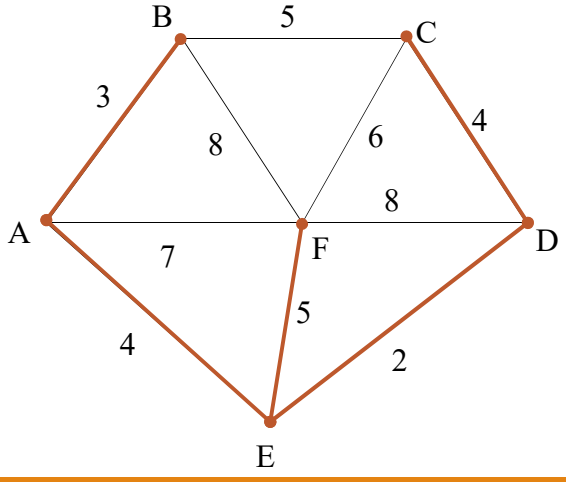
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

EF 5

Prim's Algorithm



All vertices have been connected.

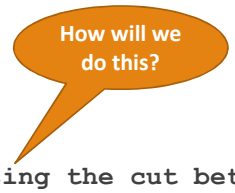
The solution is

AB 3
AE 4
ED 2
DC 4
EF 5

Total weight of tree: 18

Prim's algorithm

```
MST_PRIM(V, E)
V' = empty, E' = E
Select u from V, add u to V'
While |V'| < |V|
    select smallest edge (x,y) crossing the cut between
    |V'| and |V-V'| (x is in V')
    add y to V'
    add (x, y) to E'
return E'
```



11

We will use a "Priority queue"

A stack is first in, last out

A queue is first in, first out

A priority queue is *least-first-out*

- The "smallest" element is the first one removed
  - (You could also define a *largest-first-out* priority queue)
- The definition of "smallest" is up to the programmer
- If there are several "smallest" elements, the implementer must decide which to remove first
  - Remove any "smallest" element (don't care which)
  - Remove the first one added

12

Operations of a priority queue

Possible operations of a priority queue

- Build a priority queue
- Add an element to the priority queue
- Remove smallest element from the priority queue
- Get smallest element (without removing)
- Check if priority queue is empty
- Get the size of the priority queue

Java has a PriorityQueue class

13

Evaluating implementations

When we choose a data structure, it is important to look at usage patterns

- If we load an array once and do thousands of searches on it, we want to make searching fast—so we would probably sort the array
- If we load a huge array and expect to do only a few searches, we probably *don't* want to spend time sorting the array

For almost all uses of a queue (including a priority queue), we eventually remove everything that we add

Hence, when we analyze a priority queue, neither "add" nor "remove" is more important—we need to look at the timing for "add + remove"

14

Unsorted array implementation

A priority queue could be implemented as an *unsorted* array. If n is the number of elements in the queue, what is the complexity of adding and removing an element

- A. Adding an element - O(1), removing an element - O(n) time
- B. Adding an element - O(n), removing an element O(1)
- C. Adding an element - O(1), removing an element O(1)
- D. Adding an element - O(n), removing an element - O(n lg(n))
- E. None of the above

15

Sorted array implementations

A priority queue could be implemented as a *sorted* array. If n is the number of elements in the queue, what is the complexity of adding and removing an element?

- A. Adding an element - O(1), removing an element - O(n) time
- B. Adding an element - O(n), removing an element O(1)
- C. Adding an element - O(1), removing an element O(1)
- D. Adding an element - O(n lg(n)), removing an element - O(1)
- E. None of the above

16

Unsorted linked list implementation

A priority queue could be implemented as an *unsorted* linked list. If n is the number of elements in the queue, what is the complexity of adding and removing an element?

- A. Adding an element - O(1), removing an element - O(n) time
- B. Adding an element - O(n), removing an element O(1)
- C. Adding an element - O(1), removing an element O(1)
- D. Adding an element - O(n), removing an element - O(n lg(n))
- E. None of the above

17

Unsorted linked list implementation

A priority queue could be implemented as an *sorted* linked list. If n is the number of elements in the queue, what is the complexity of adding and removing an element?

A priority queue could be implemented as a *sorted* linked list

- A. Adding an element - O(1), removing an element - O(n) time
- B. Adding an element - O(n), removing an element O(1)
- C. Adding an element - O(1), removing an element O(1)
- D. Adding an element - O(n), removing an element - O(n lg(n))
- E. None of the above

18

Unbalanced binary tree implementation

A priority queue could be represented as a (not necessarily balanced) binary search tree

- A. Adding an element - O(log n) removing an element - O(n)
- B. Adding an element ranges from O(log n) to O(n), removing an element ranges O(log n) to O(n)
- C. Adding an element - O(log n), removing an element - O(log n)
- D. Adding an element - O(n log n), removing an element – O (1)
- E. Adding an element - O (1), removing an element - O(log n)

19

Binary tree implementations

A priority queue could be represented as a *balanced* binary search tree

- Insertion and removal could destroy the balance
- We need an algorithm to *rebalance* the binary tree
- Good rebalancing algorithms require only O(log n) time, and are complicated

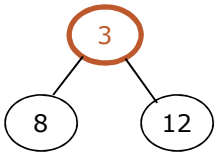
20

Heap implementation

A priority queue can be implemented as a heap

In order to do this, we have to define the *heap property*

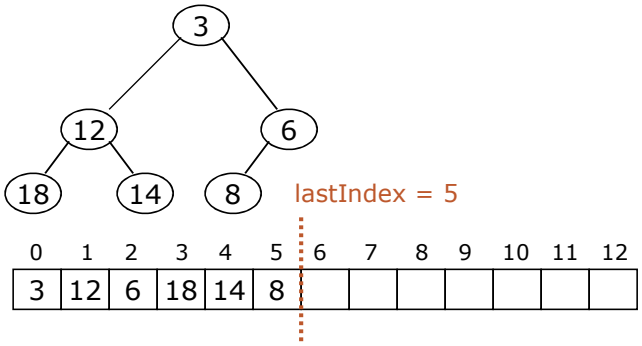
For a priority queue, we will define a node to have the heap property if it is *as least as small* as its children (since we are using smaller numbers to represent higher priorities)



Priority queue: Orange node has the heap property

21

Array representation of a heap



Left child of node i is 2*i + 1, right child is 2*i + 2

- Unless the computation yields a value larger than lastIndex, in which case there is no such child

Parent of node i is (i − 1)/2

- Unless i == 0

Using the heap

To add an element:

- Increase lastIndex and put the new value there
- Reheap the newly added node by swapping with parent node, until heap property is restored
  - This is called up-heap bubbling or percolating up
  - Up-heap bubbling requires O(log n) time

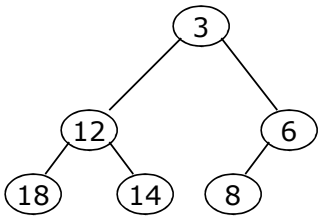
To remove an element:

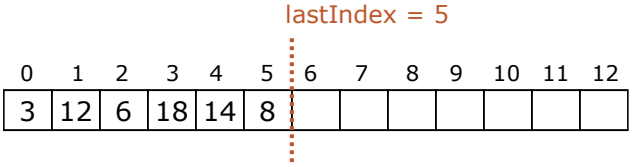
- Remove the element at location 0
- Move the element at location lastIndex to location 0, and decrement lastIndex
- Reheap the new root node (the one now at location 0) by swapping with smallest child element until heap property is restored
  - This is called down-heap bubbling or percolating down
  - Down-heap bubbling requires O(log n) time

Thus, it requires O(log n) time to add *and* remove an element

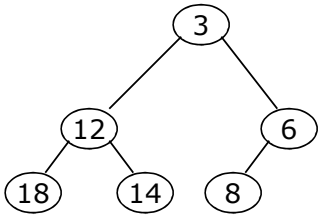
23

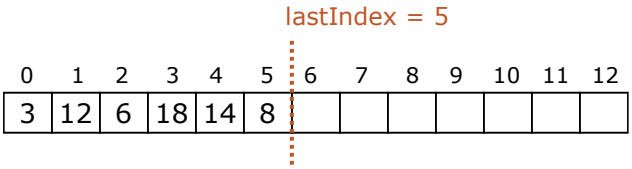
Example: up-heap (percolate up)





Example: down-heap (percolate down)





Comments

A priority queue is a data structure that is designed to return elements in order of priority

Efficiency is usually measured as the *sum* of the time it takes to add and to remove an element

Simple implementations take O(n) time

A heap implementation takes O(log n) time

Thus, for any sort of heavy-duty use, a heap implementation is better

26

Prim's algorithm with priority queue

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

Proof or Fluff

Let T=(V, E) be a tree. Prove that |E|=|V|-1

Consider the following proof by induction on V:

Base case: Clearly, this is true for |V|=1 and |V|=2.

Inductive hypothesis: suppose true for trees with |V|-1 vertices. Then this tree has |V|-1-1 edges. We can construct another tree by adding one new vertex and connecting it to one of vertices in the tree with one edge. Thus, we have a tree with |V| vertices and |V|-1-1+1= |V|-1 edges. This proves that |E|=|V|-1.

A. This proof is valid

B. This proof is flawed because it proves that there exists a tree with |E|=|V|-1 and not the general case

C. This proof is flawed because it doesn't prove that the newly constructed tree is in fact a tree.

D. This proof is flawed for some other reason

Let T=(V, E) be a tree. Prove that |E|=|V|-1

Consider the following proof by induction on V:

Base case: Clearly, this is true for |V|=1 and |V|=2.

Inductive hypothesis: suppose true for trees with n<|V| vertices.

Let T be a tree with |V| vertices. Let e be an edge connecting vertices u and v in T. Since T is a tree, there is a unique path from u to v and it has to be via edge e. If we remove e, T will become disconnected. Now T-{e} consists of two components T_1 and T_2 and those components are trees (since there were no cycles in T to begin with).

Let n_1 be the number of vertices in T_1 and n_2 be the number of vertices in T_2, so $n_1+n_2=|V|$.

Also $0 < n_1 < |V|$ and $0 < n_2 < |V|$. By inductive hypothesis the number of edges in T_1 is n_1-1 and the number of edges in T_2 is n_2-1. Thus, the number of edges in T is $n_1-1+n_2-1+1=n_1+n_2-1=|V|-1$.

A. This proof is valid

B. This proof is flawed because $n_1+n_2=|V|$ is false

C. This proof is flawed because T_1 and T_2 are not guaranteed to be trees.

D. This proof is flawed for some other reason