Graphs – Dijkstra's, Prim's, Indirect Heaps

CS4102, Spring 2021

Readings: CLRS 23.2, 24.2, 24.3

Topics

- Dijkstra's algorithm + naïve runtime
  - Review!!
- Prim's algorithm + naïve runtime
  - Also Review!!!
- Why these two algorithms? Turns out they are VERY similar
- Indirect Heaps
  - A new data structure that makes both algorithms above more efficient

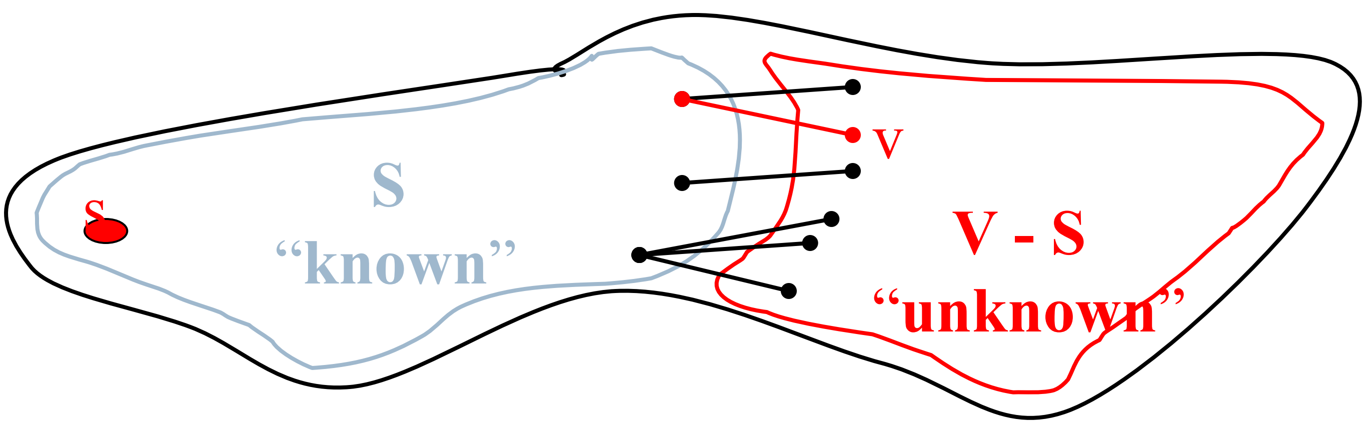
Dijkstra’s Algorithm

Weighted Shortest Path

- no negative weight edges.
- **Dijkstra's algorithm**: uses similar ideas as the unweighted case.

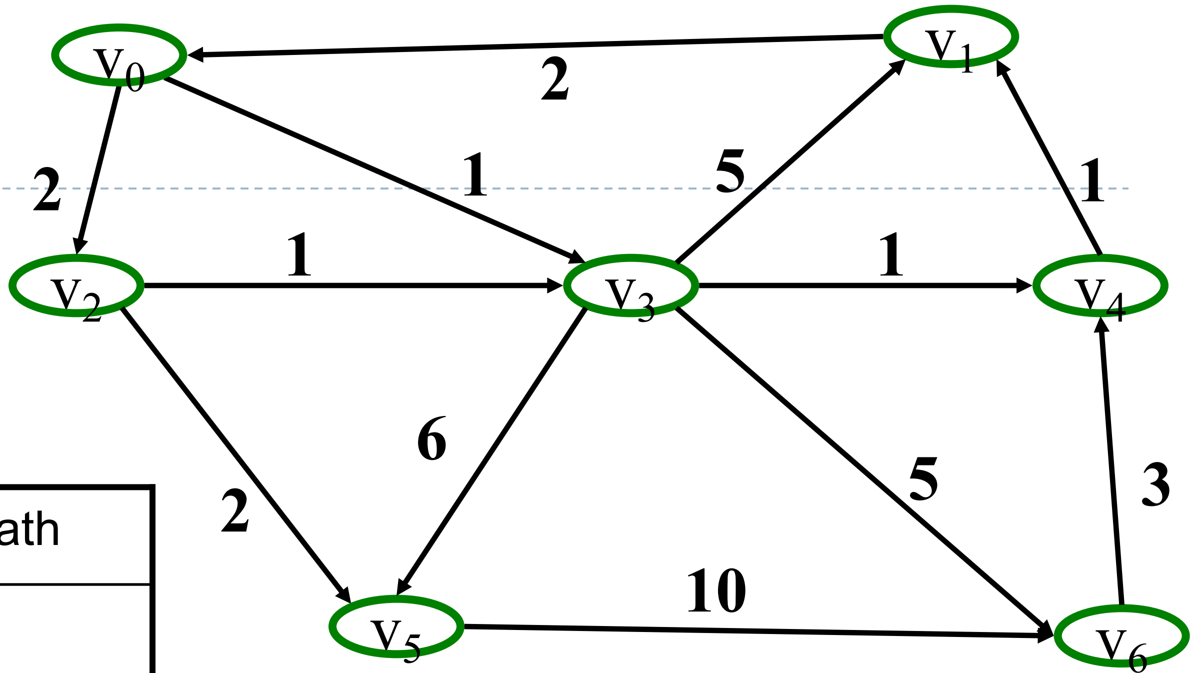
Greedy algorithms:

do what seems to be best at every decision point.



Dijkstra's algorithm

- Initialize each vertex's distance as infinity
- Start at a given vertex *s*
  - Update *s*'s distance to be 0
- Repeat
  - Pick the next unknown vertex with the shortest distance to be the next *v*
    - If no more vertices are unknown, terminate loop
  - Mark *v* as known
  - For each edge from *v* to adjacent unknown vertices *w*
    - If the total distance to *w* is less than the current distance to *w*
      - Update *w*'s distance and the path to *w*



V	Known	Dist	path
v0			
v1			
v2			
v3			
v4			
v5			
v6			

```
void Graph::dijkstra(Vertex s){
  Vertex v,w;
  s.dist = 0;

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

    for each w adjacent to v
      if (!w.known)
        if (v.dist + Cost_VW < w.dist){
          w.dist = v.dist + Cost_VW;
          w.path = v;
        }
  }
}
```

Naïve Analysis

- How long does it take to find the smallest unknown distance?
  - simple scan using an array: $O(v)$
- Total running time:
  - Using a simple scan: $O(v^2+e) = O(v^2)$

Dijkstra' Algorithm

```
dijkstra(G, wt, s)
 init PQ to be empty;
 PQ.Insert(s, dist=0);
 parent[s] = NULL; dist[s] = 0;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        dist[w] = dist[v] + wt(v,w)
        PQ.Insert(w, dist[w] );
        parent[w] = v;
     }
     else if (w is fringe && dist[v] + wt(v,w) <
  dist[w] ) {
        dist[w] = dist[v] + wt(v,w)
        PQ.decreaseKey(w, dist[w]);
        parent[w] = v;
     }
```

Analysis of Priority Queue implementation?

- How long does it take to find the smallest unknown distance?
  - extract min from PQ: O(log(V))
  - But called V times total, so O(V*log(V))

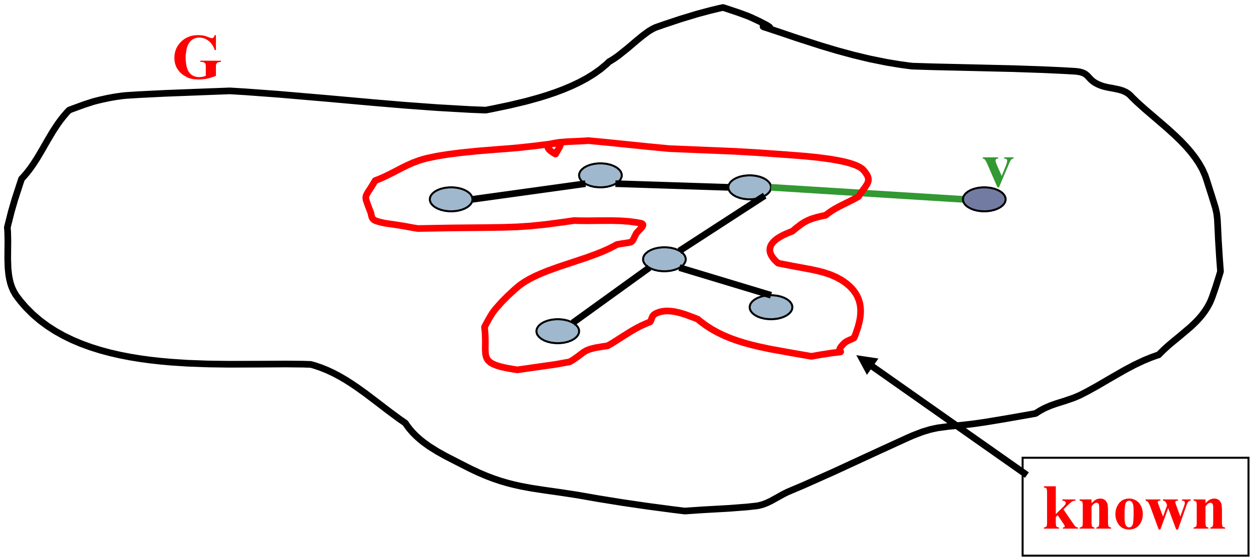
- Inner loop:
  - runs E times like before but….
  - Each edge could force a PQ.decreaseKey() call, runtime??
  - Naïve decreaseKey() is linear time: O(V), total of O(E*V)

- So, total is O(V*log(V) + E*V). Is this better??

Prim’s Algorithm

Prim's algorithm

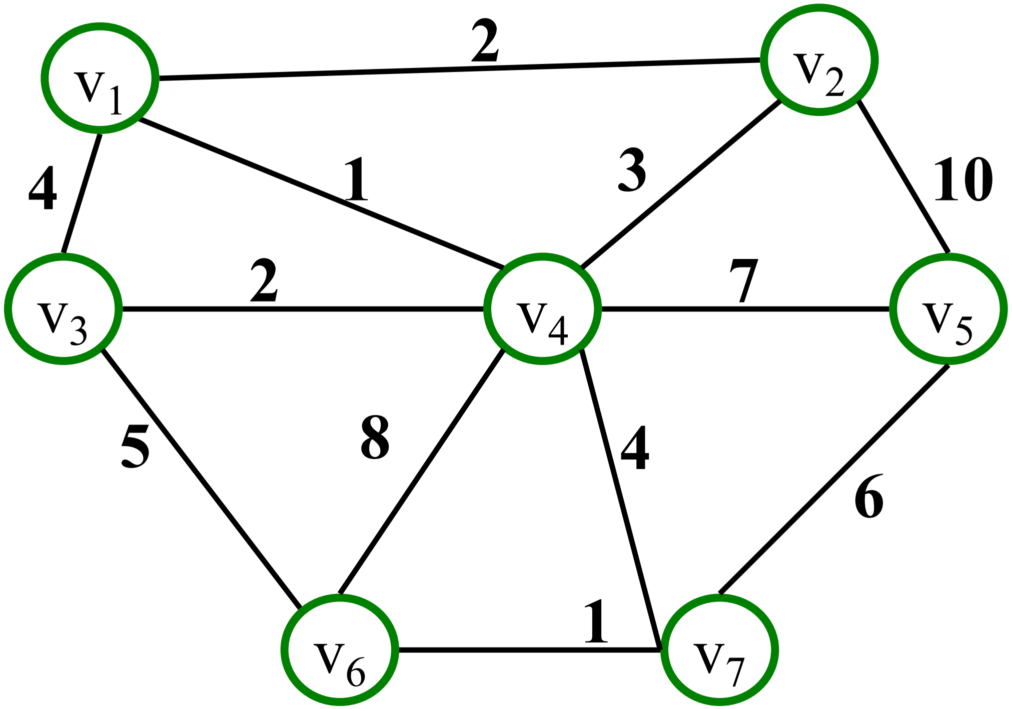
Idea: Grow a tree by adding an edge from the "known" vertices to the "unknown" vertices. Pick the edge with the smallest weight.



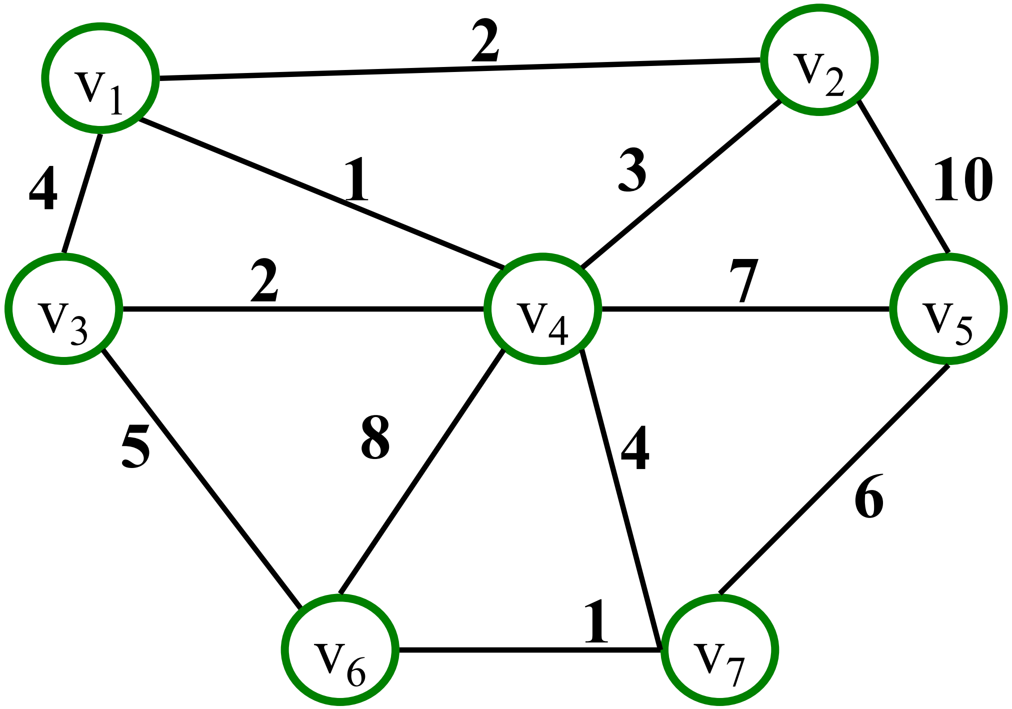
Prim’s Algorithm for MST

- Pick one node as the root,
- Incrementally add edges that connect a “new” vertex to the tree.
- Pick the edge (u,v) where:
  - u is in the tree, v is not AND
  - where the edge weight is the smallest of all edges (where u is in the tree and v is not).

MST



MST



v1

{v1, v4}

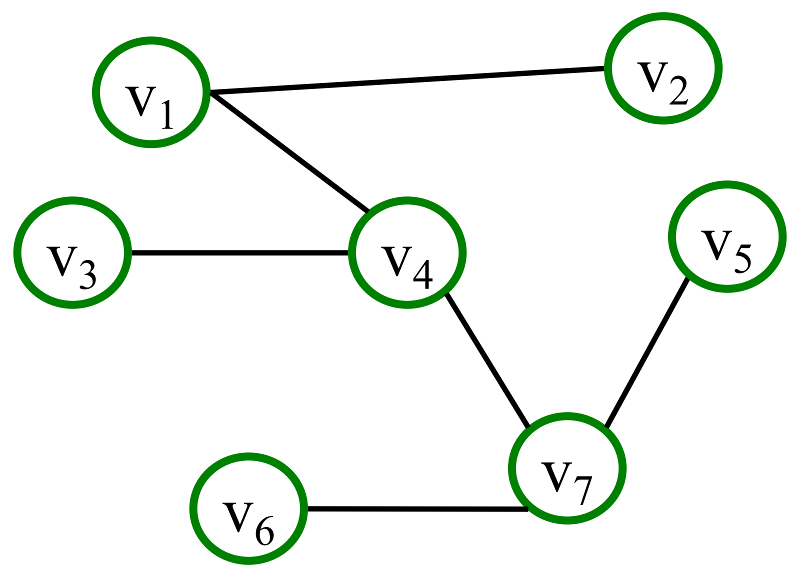
{v1, v2}

{v4, v3}

{v4, v7}

{v7, v6}

{v7, v5}



Prim's MST Algorithm

- Greedy strategy:
  - Choose some start vertex as current-tree
  - Greedy rule: Add edge from graph to current-tree that
    - has the lowest weight of edges that…
    - have one vertex in the tree and one not in the tree.
- Thus builds-up one tree by adding a new edge to it
- Can this lead to an infeasible solution? (Tell me why not.)
- Is it optimal? (Yes. Need a proof.)

Tracking Edges for Prim's MST

- Candidates edges: edge from a tree-node to a non-tree node
  - Since we'll choose smallest, keep only one candidate edge for each non-tree node
  - But, may need to make sure we always have the smallest edge for each non-tree node
- Fringe-nodes: non-trees nodes adjacent to the tree
- Need data structure to hold fringe-nodes
  - Priority queue, ordered by min-edge weight
  - May need to update priorities!

Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```

Cost of Prim's Algorithm

- Looks VERY similar to Dijkstra's doesn't it!!

- Outer loop extracts from PQ total of V times
  - O(V*log(V))

- Inner loop runs E times total, but calls decreaseKey()
  - If decreaseKey() is naïve and linear (V), then
  - O(E*V)

- Total: O(V*log(V) + E*V)

Indirect Heaps

Compare

- Both Dijkstra and Prim have same structure, and suffer from a naïve, slow implementation of decreaseKey()

- Let's compare the code real fast, and then introduce the ***Indirect Heap***

Dijkstra' Algorithm

```
dijkstra(G, wt, s)
 init PQ to be empty;
 PQ.Insert(s, dist=0);
 parent[s] = NULL; dist[s] = 0;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        dist[w] = dist[v] + wt(v,w)
        PQ.Insert(w, dist[w] );
        parent[w] = v;
     }
     else if (w is fringe && dist[v] + wt(v,w) <
  dist[w] ) {
        dist[w] = dist[v] + wt(v,w)
        PQ.decreaseKey(w, dist[w]);
        parent[w] = v;
     }
```

Prim’s Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```

Better PQ Implementations

- Goal: Lower cost of PQ.decreaseKey()

- Example of naïve approach first →

Better PQ Implementations

- Goal: Lower cost of PQ.decreaseKey()

- Indirect Heap →

Better PQ Implementations (2)

- Cost of Dijkstra's and Prim's
  - O(V*log(V) + E***V**)

- Indirect heap makes bolded V become log(V)

- New Cost:
  - O(V*log(V) + E*log(V)) = O(E*log(V))

Summary

What Did We Learn?

- Review of Dijkstra's and Prim's
  - Almost same algorithm but solve different problems!!
- Review of Naïve runtime analysis
- Indirect heap and better runtime for each algorithm