ECE430.217 Data Structures

Prim's Algorithm

Textbook: Weiss Chapter 9.5

Byoungyoung Lee
https://compsec.snu.ac.kr
byoungyoung@snu.ac.kr

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

Outline

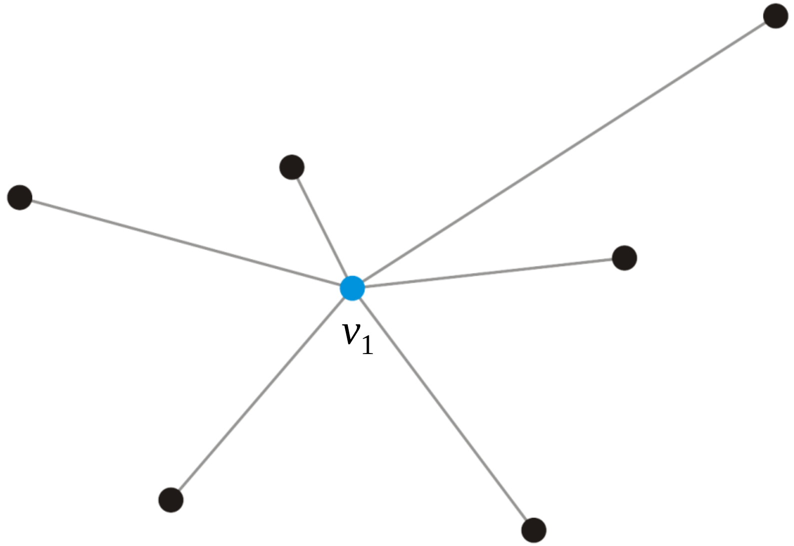
This topic covers Prim's algorithm:

- Finding a minimum spanning tree (MST)
- The idea and the algorithm
- An example

Observation

Suppose we take a vertex

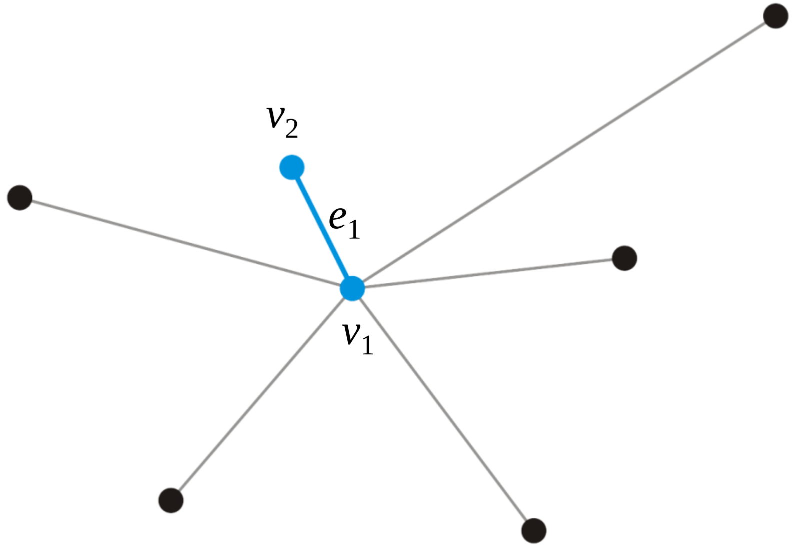
– Given a single vertex v_1, it forms a minimum spanning tree (MST) on one vertex



Observation

Add the adjacent vertex v_2 that has a connecting edge e_1 of minimum weight

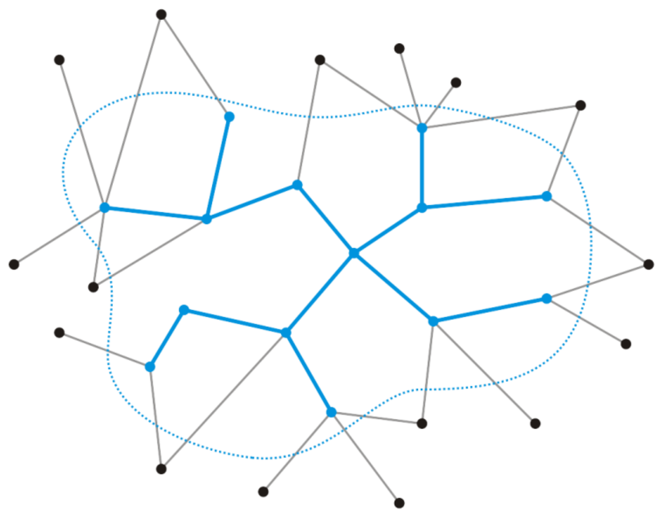
- This forms a MST on these two vertices



Strategy

Strategy:

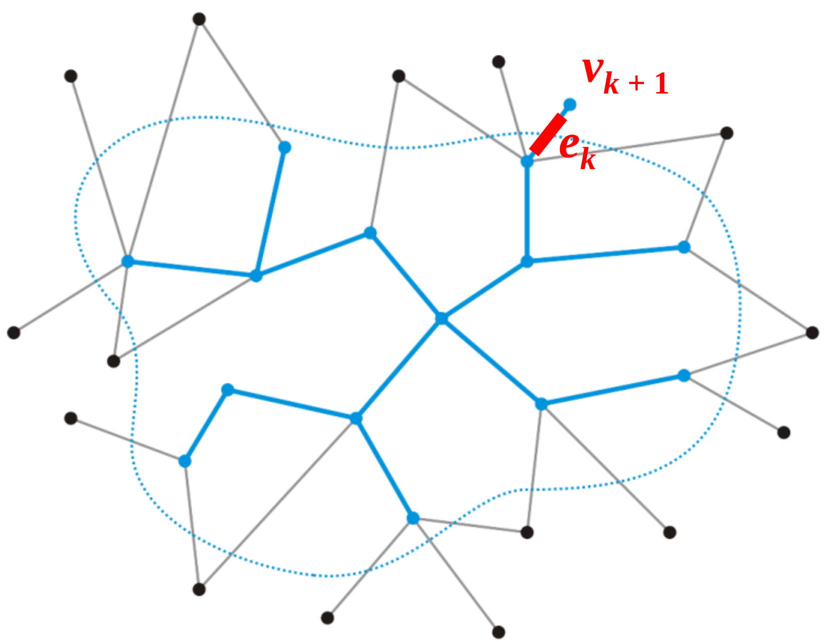
- Suppose we have a known MST on $k < n$ vertices
- How could we extend this MST?



Strategy

Suppose you add $e_{k,}$,which has the minimum weight out of all edges connected to this MST

- Adding e_k does create an MST with $k+1$ nodes to connect v_{k+1}
  - Given the current MST, no lighter edges would connect to v_{k+1}
- However, can any edge other than e_k be used to connect v_{k+1} in an MST with n nodes later?

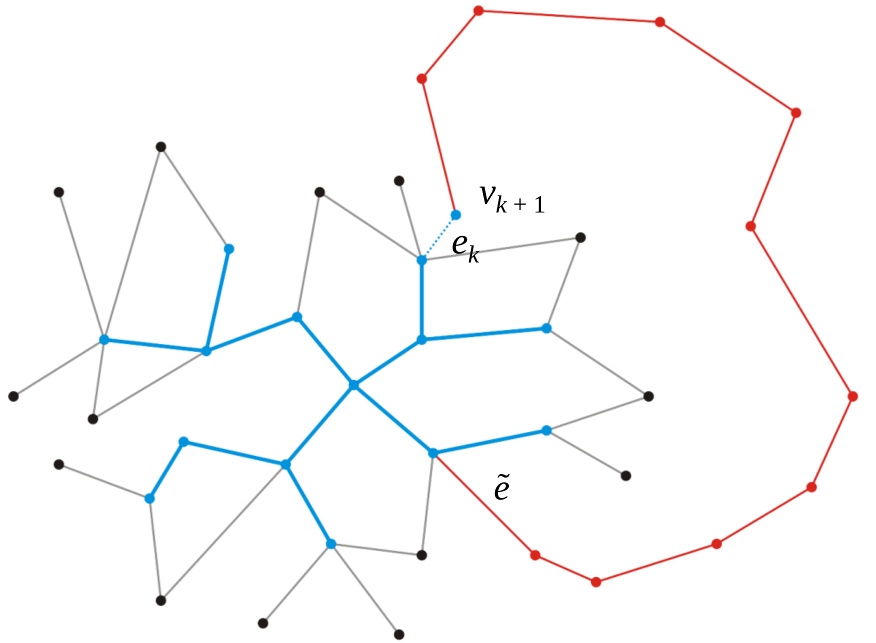


Proof by Contradiction

Proof by contradiction:

Assume the previous claim is false.

- Thus, vertex v_{k+1} is connected to the MST via another sequence of edges
- Out of such sequence of edges, let's call $\tilde{e}$ as the edge out connecting to the existing MST



Proof by Contradiction

Let w be the weight of this MST (with $\tilde{e}$)

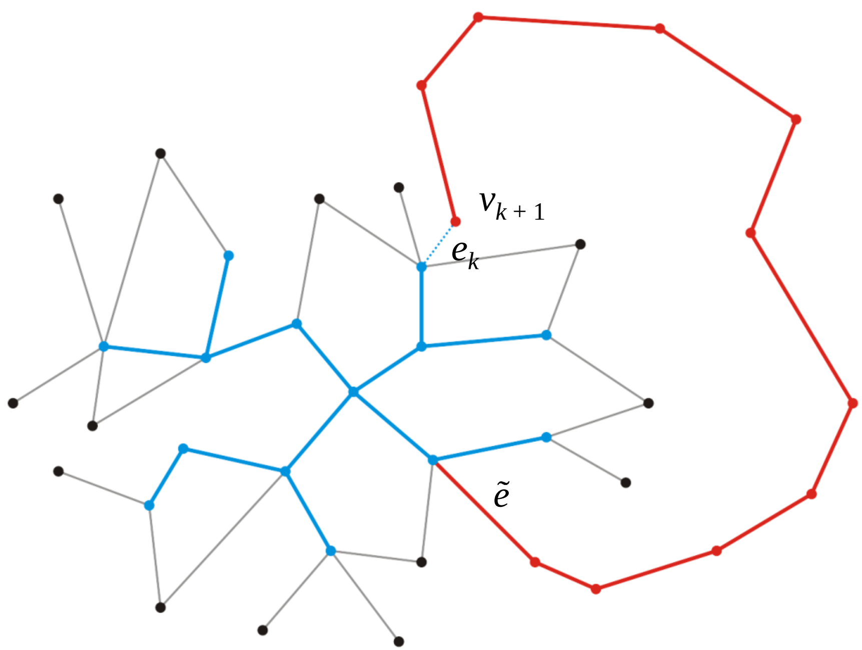
- Recall that we picked e_k because $|\tilde{e}| > |e_k|$
  - $|e|$ denotes the weight of the edge e

Suppose we add e_k and exclude $\tilde{e}$ to the MST

- The result is still a spanning tree, but the weight is now
  - $w + |e_k| - |\tilde{e}| \leq w$

This contradicts our assumption that the MST with $\tilde{e}$ is minimal

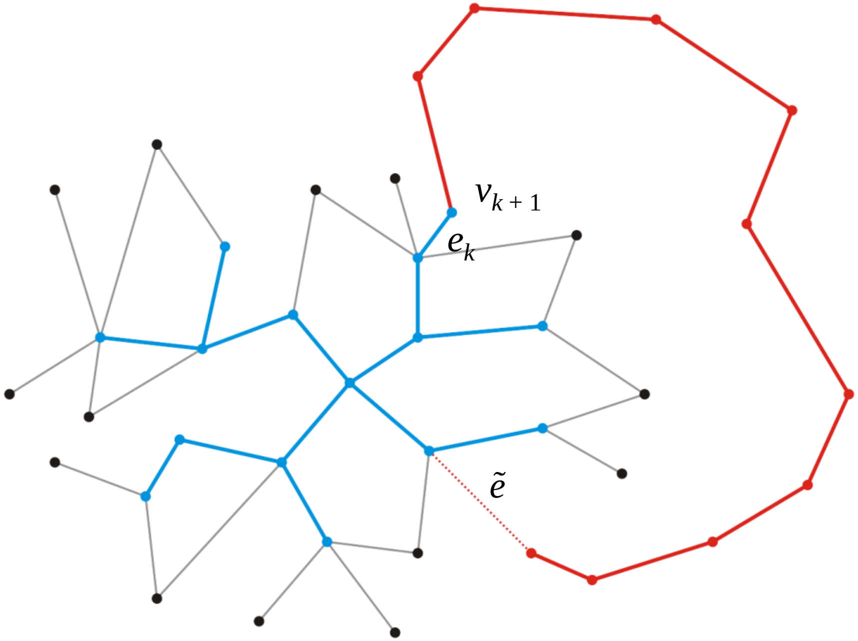
Therefore, our MST must contain e_k



From Strategy to Prim's Algorithm

We keep this strategy for all vertices, starting with k=1

➔ Prim's algorithm



Prim's Algorithm

Prim's algorithm for finding the MST:

- Start with an arbitrary vertex to form a MST on one vertex
- At each step, add the vertex v not yet in the MST
  - Vertex v is connected with an edge with least weight to the existing minimum spanning sub-tree
- Continue until we have $n - 1$ edges and n vertices

Note: Prim's algorithm is **a greedy algorithm**

➔ A greedy algorithm does not always yield the optimal solution

Q. Does Prim's algorithm guarantee the MST
(i.e., the optimal solution)?

Prim’s Algorithm: Data Structures

Associate each vertex with:

- A Boolean flag indicating if the vertex has been visited,
- The minimum distance to the partially constructed MST, and
- A pointer to that vertex which will form the parent node in the resulting tree

Prim’s Algorithm: Initialization

Initialization:

- Select a root node and set its distance as 0
- Set the distance to all other vertices as ∞
- Set all vertices to being unvisited
- Set the parent pointer of all vertices to 0

Iterate while there exists an unvisited vertex with distance < ∞

- Select that unvisited vertex with minimum distance
- Mark that vertex as having been visited
- For each adjacent vertex, if the weight of the connecting edge is less than the current distance to that vertex:
  - Update the distance to be the weight of the connecting edge
  - Set the current vertex as the parent of the adjacent vertex

Prim’s Algorithm: Halting Condition

Halting Conditions:

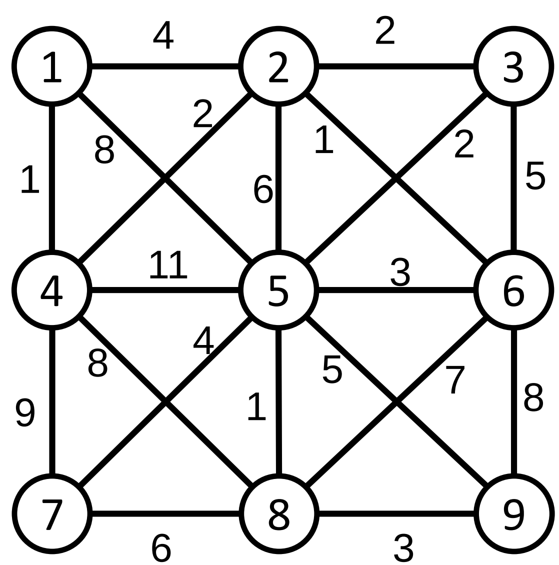
- There are no unvisited vertices which have a distance < ∞

If all vertices have been visited, we have a spanning tree of the entire graph

If at any point, all remaining vertices had a distance of ∞, this indicates that the graph is not connected ➔ No MST

Prim’s Algorithm: Example

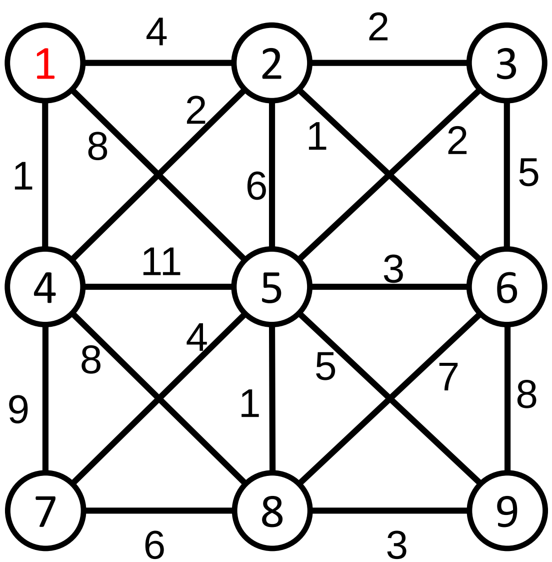
Let us find the minimum spanning tree for the following undirected weighted graph



Prim's Algorithm: Example

First we set up the appropriate table and initialize it

- Suppose the root is the vertex 1

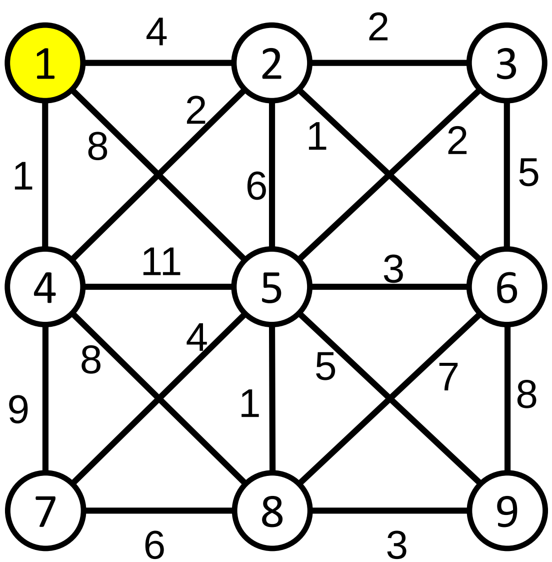


		Distance	Parent
1	F	0	0
2	F	∞	0
3	F	∞	0
4	F	∞	0
5	F	∞	0
6	F	∞	0
7	F	∞	0
8	F	∞	0
9	F	∞	0

Prim's Algorithm: Example

Visit vertex 1

- We update vertices 2, 4, and 5
- MST: {1}

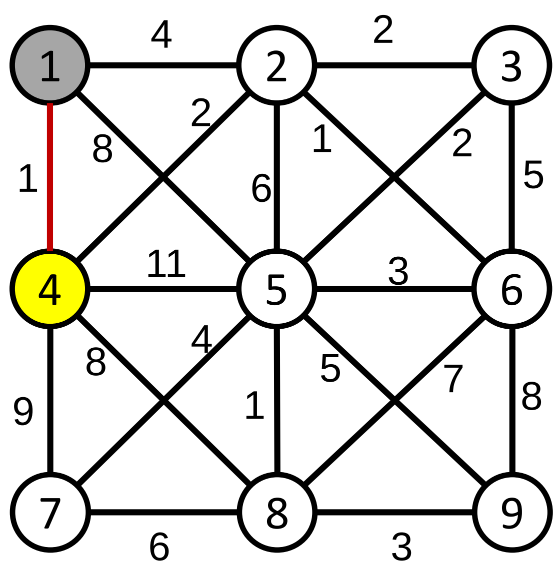


		Distance	Parent
1	F➔T	0	0
2	F	∞➔4	0➔1
3	F	∞	0
4	F	∞➔1	0➔1
5	F	∞➔8	0➔1
6	F	∞	0
7	F	∞	0
8	F	∞	0
9	F	∞	0

Prim's Algorithm: Example

Visit vertex 4, because vertex 4 has the minimum distance (among unvisited vertices)

- Update vertices 2, 7, 8
- Don't update vertex 5
- MST: {1,4}

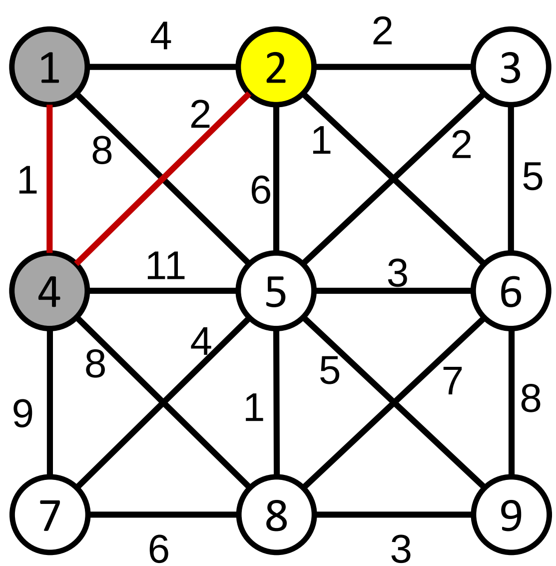


		Distance	Parent
1	T	0	0
2	F	4→2	1→4
3	F	∞	0
4	F→T	1	1
5	F	8	1
6	F	∞	0
7	F	∞→9	0→4
8	F	∞→8	0→4
9	F	∞	0

Prim’s Algorithm: Example

Visit vertex 2

- Update 3, 5, and 6
- MST: {1, 4, 2}

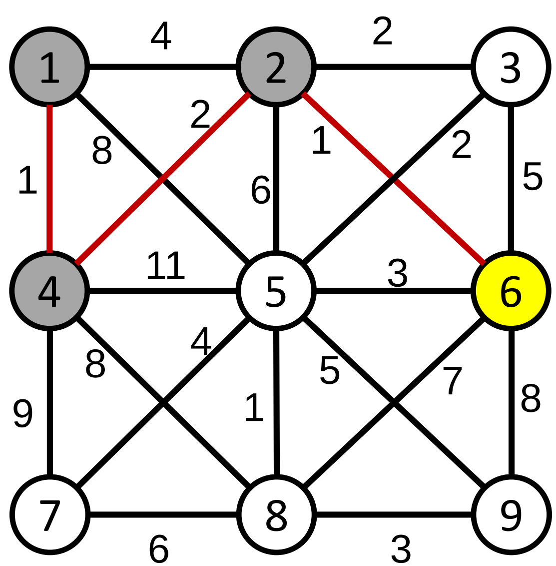


		Distance	Parent
1	T	0	0
2	F→T	2	4
3	F	∞→2	0→2
4	T	1	1
5	F	8→6	1→2
6	F	∞→1	0→2
7	F	9	4
8	F	8	4
9	F	∞	0

Prim’s Algorithm: Example

Next, we visit vertex 6:

- update vertices 5, 8, and 9
- MST: {1, 4, 2, 6}

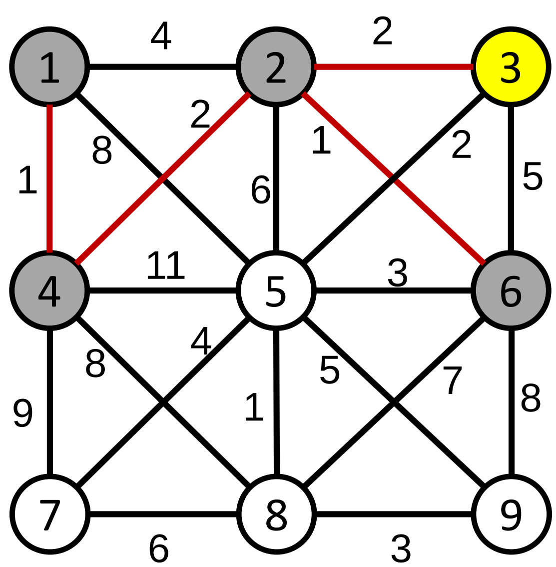


		Distance	Parent
1	T	0	0
2	T	2	4
3	F	2	2
4	T	1	1
5	F	6→3	2→6
6	F→T	1	2
7	F	9	4
8	F	8→7	4→6
9	F	∞→8	0→6

Prim’s Algorithm: Example

Visit vertex 3, and update vertex 5

- MST: {1, 4, 2, 6, 3}

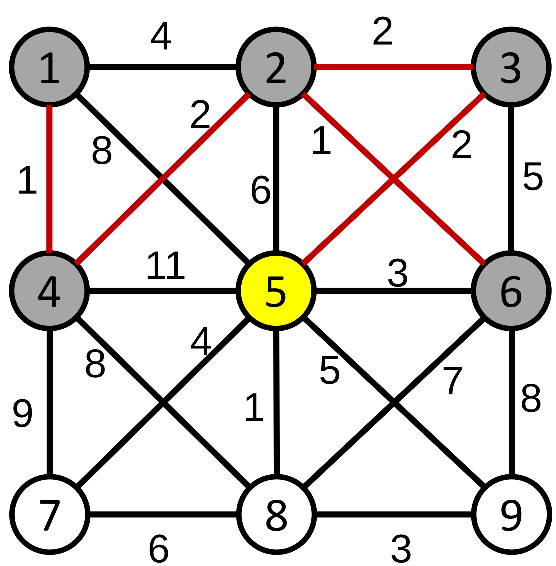


		Distance	Parent
1	T	0	0
2	T	2	4
3	F→T	2	2
4	T	1	1
5	F	3→2	6→3
6	T	1	2
7	F	9	4
8	F	7	6
9	F	8	6

Prim’s Algorithm: Example

Visit vertex 5

- No need to update other vertices
- MST: {1, 4, 2, 6, 3, 5}

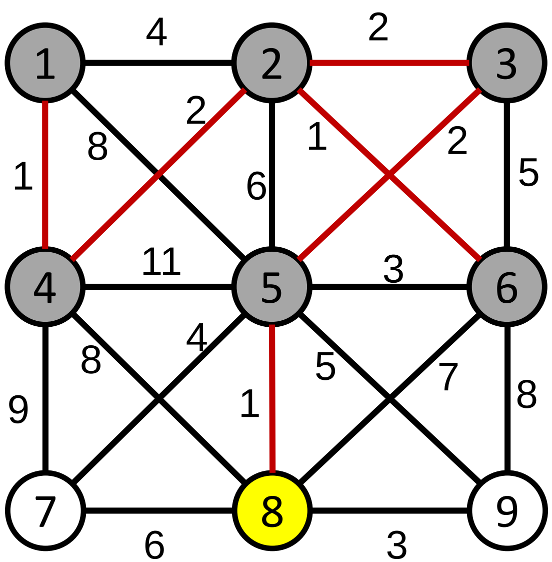


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	F➔T	2	3
6	T	1	2
7	F	9	4
8	F	7	6
9	F	8	6

Prim's Algorithm: Example

Visiting vertex 8, we only update vertex 9

- MST: {1, 4, 2, 6, 3, 5, 8}

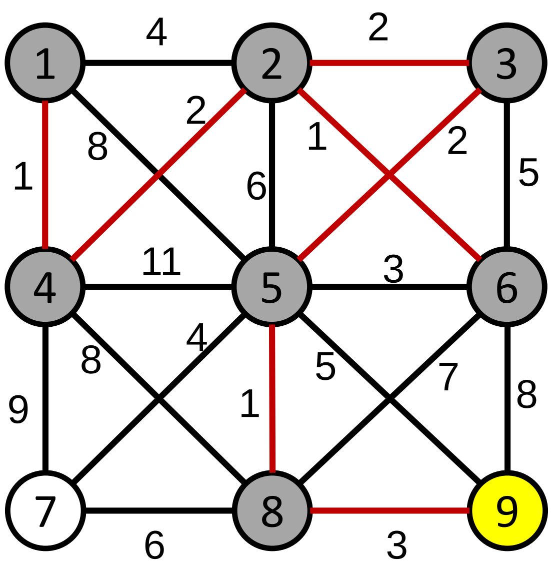


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	F→T	1	5
9	F	5→3	5→8

Prim's Algorithm: Example

Visit vertex 9. No need to update other vertices.

- MST: {1, 4, 2, 6, 3, 5, 8, 9}

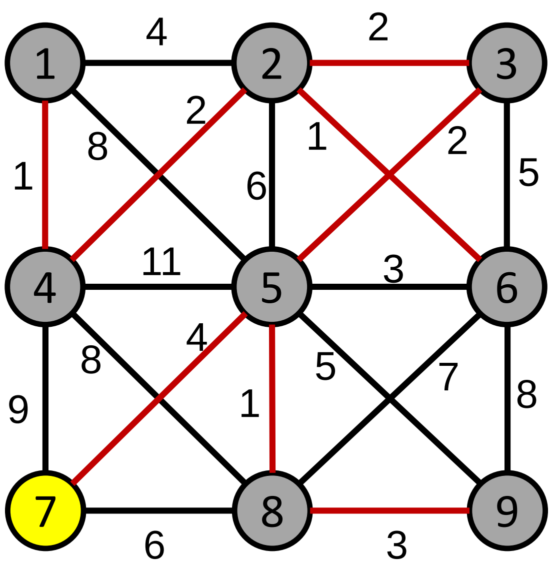


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	T	1	5
9	F➔T	3	8

Prim's Algorithm: Example

Visit vertex 7, then done.

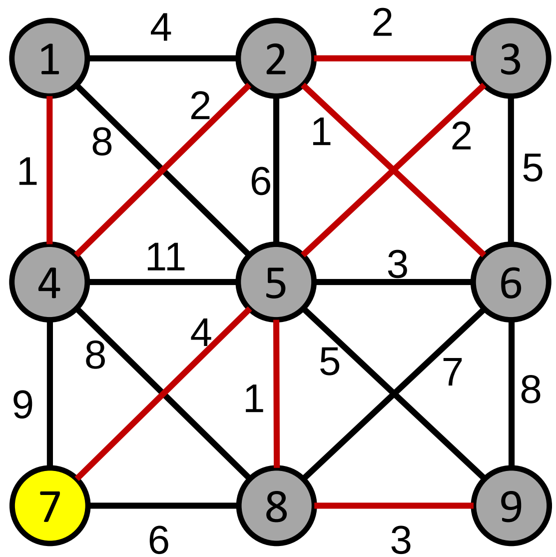
- MST: {1, 4, 2, 6, 3, 5, 8, 9, 7}



		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F➔T	4	5
8	T	1	5
9	T	3	8

Prim's Algorithm: Example

Using the parent pointers, we can now construct the minimum spanning tree



		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	T	4	5
8	T	1	5
9	T	3	8

Implementation and analysis

The initialization requires $\Theta(|V|)$ memory and run time

Iteration: We iterate $|V| - 1$ times, each time finding the *min. distance* vertex

- Iterating through the table (to find the min. distance vertex) requires is $\Theta(|V|)$ time
- Each time we find a min. distance vertex, we must check all of its neighbors (to update distance)

```
for _ in range(|V|): // until visiting all vertices
    // Find the min distance vertex (among unvisited ones)
    v = table.find_min_dist_vertex()
    table.mark_visit(v)
    // Visit: update the distance if needed
    for j in graph.get_adj_vertices(v):
        if (graph.get_dist(v, j) < table.get_dist(j))
            table.set_dist(j, graph.get_dist(v, j))
```

With an adjacency matrix, the run time is $O(|V|(|V| + |V|)) = O(|V|^2)$

- Each call of find_min_dist_vertex() takes $O(|V|)$
- Enumerating adj vertices for each vertex take $O(|V|)$

With an adjacency list, the run time is $O(|V|^2 + |E|) = O(|V|^2)$ as $|E| = O(|V|^2)$

- Each call of find_min_dist_vertex() takes $O(|V|)$
- Enumerating all adj vertices in the end would take $O(|E|)$ for all enumerations

Implementation and analysis

Can we do better?

- Recall, we only need the next shortest edge
- How about a priority queue?
  - Assume we are using a binary heap
  - We will have to update the heap structure

Implementation and analysis: Binary Heap

```
for _ in range(|V|): // until visiting all vertices
    // Find the min distance vertex (among unvisited ones)
    v = table.find_min_dist_vertex()
    table.mark_visit(v)
    // update the distance if needed
    for j in graph.get_adj_vertices(v):
        if (graph.get_dist(v, j) < table.get_dist(j))
            table.set_dist(j, graph.get_dist(v, j))
```

The table is maintained with a min heap, where the key is a min. distance associated with vertex

- find_min_dist_vertex() takes $\ln(|V|)$, which is executed $|V|$ times
- table.set_dist() takes $\ln(|V|)$, which is executed $|E|$ times

Thus, the total run time with binary heap is

$$O(|V| \ln(|V|) + |E| \ln(|V|)) = O(|E| \ln(|V|))$$

Summary

We have seen an algorithm for finding minimum spanning trees
- Start with a trivial minimum spanning tree and grow it
- An alternate algorithm, Kruskal's algorithm, uses a different approach

Prim's algorithm finds an edge with least weight which grows an already existing tree

References

Wikipedia, http://en.wikipedia.org/wiki/Minimum_spanning_tree
Wikipedia, http://en.wikipedia.org/wiki/Prim’s_algorithm