# MA/CSSE 473 Day 33

Student Questions

Change to HW 13

Minimal Spanning Tree

Kruskal

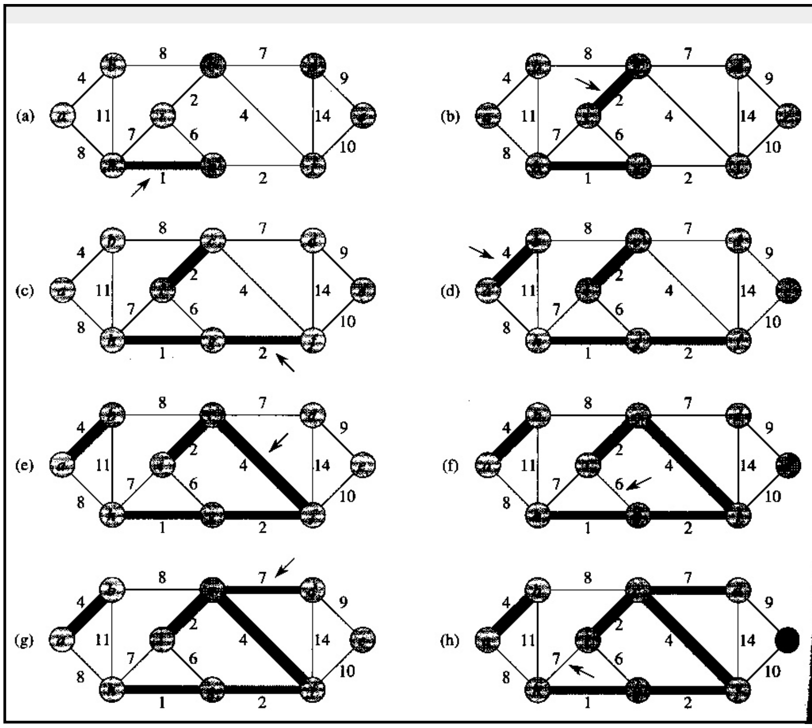
Prim

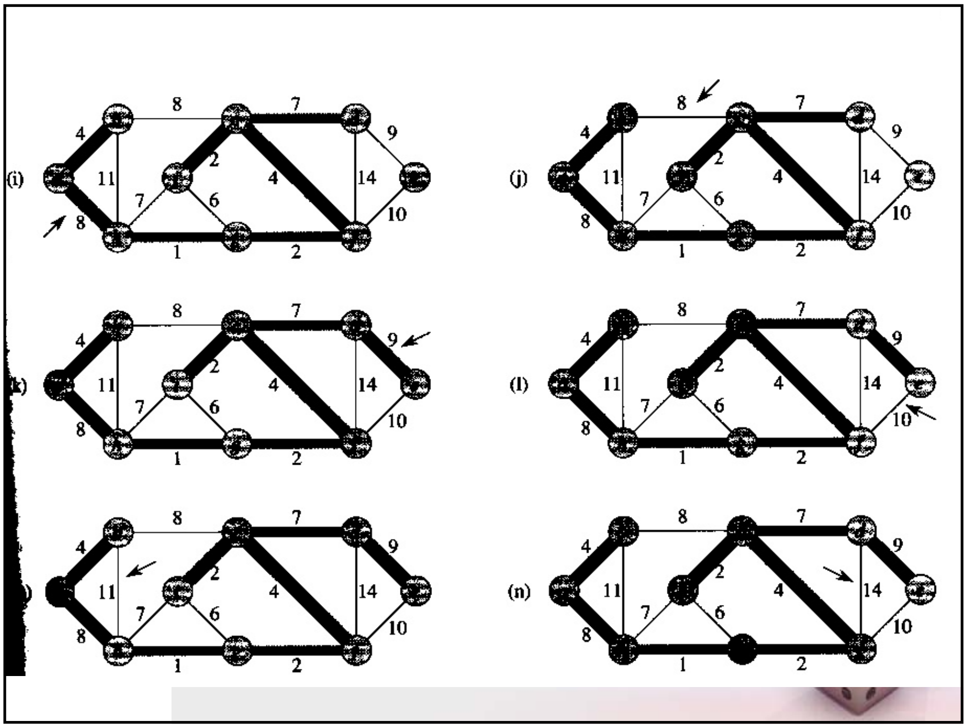
Kruskal and Prim

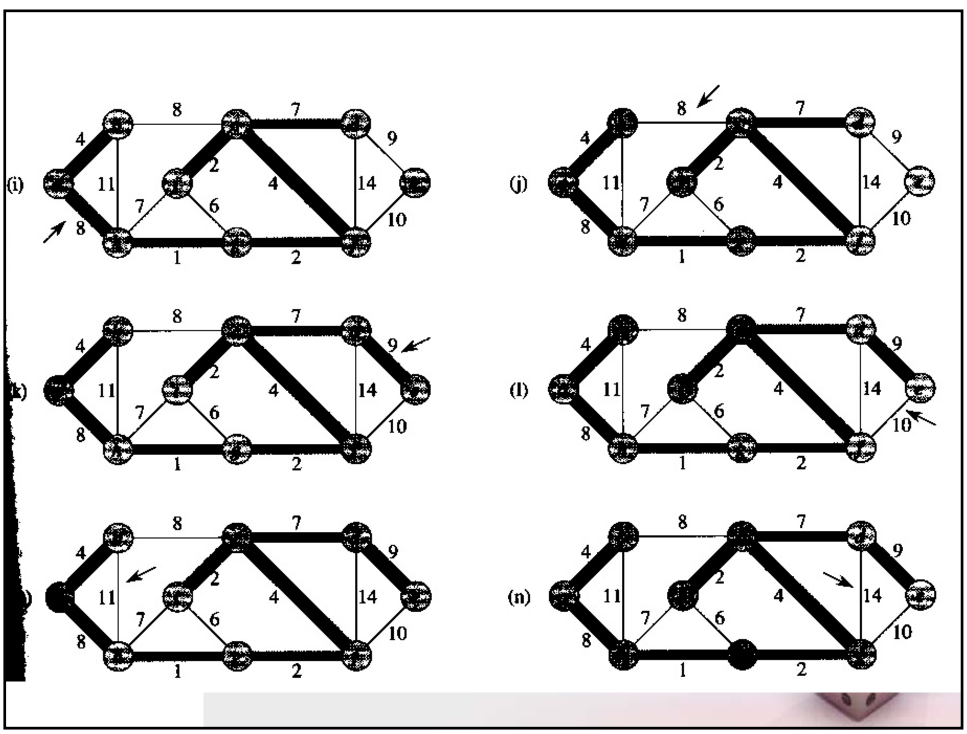
## ALGORITHMS FOR FINDING A MINIMAL SPANNING TREE

# Kruskal’s algorithm

- To find a MST (minimal Spanning Tree):
- Start with a graph T containing all n of G’s vertices and none of its edges.
- for i = 1 to n – 1:
  - Among all of G’s edges that can be added without creating a cycle, add to T an edge that has minimal weight.
  - Details of Data Structures later

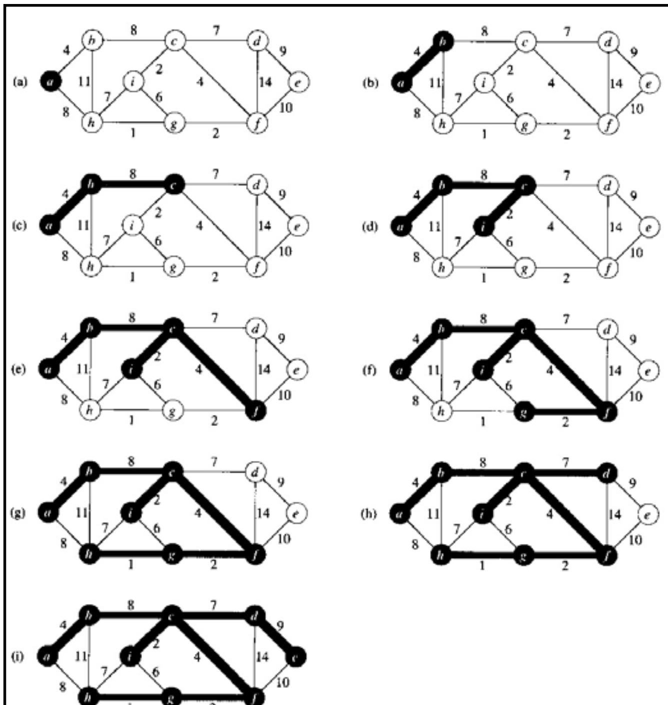



(i)
(j)
(k)
(l)
(n)


(i)
(j)
(k)
(l)
(n)

# Prim's algorithm

- Start with T as a single vertex of G (which *is* a MST for a single-node graph).
- for i = 1 to n – 1:
  - Among all edges of G that connect a vertex in T to a vertex that is not yet in T, add a minimum-weight edge (and the vertex at the other end of T).
  - Details of Data Structures later




**Example of Prim's algorithm**

# Correct?

- These algorithms seem simple enough, but do they really produce a MST?
- We examine  lemma that is the crux of both proofs.
- It is subtle, but once we have it, the proofs are fairly simple.

# MST lemma

- Let G be a weighted connected graph,
- let T be any MST of G,
- let G′ be any subgraph of T, and
- let C be any connected component of G′.
- Then:
  - If we add to C an edge *e=(v,w)* that has minimum-weight among all edges that have one vertex in C and the other vertex not in C,
  - G has an MST that contains the union of G′ and *e*.

[WLOG, v is the vertex of e that is in C, and w is not in C]

**Summary:** If G' is a subgraph of an MST, so is G'∪{e}

**Q2**

## Recap: MST lemma

Let G be a weighted connected graph with an MST T;
let G′ be any subgraph of T, and let C be any connected component of G′.

If we add to C an edge *e=(v,w)* that has minimum-weight among all edges that have one vertex in C and the other vertex not in C,

then G has an MST that contains the union of G′ and *e*.

## Recall Kruskal's algorithm

- To find a MST for G:
  - Start with a connected weighted graph containing all of G's n vertices and none of its edges.
  - for i = 1 to n – 1:
    - Among all of G's edges that can be added without creating a cycle, add one that has minimal weight.

**Does this algorithm actually produce an MST for G?**

## Does Kruskal produce a MST?

- **Claim:** After every step of Kruskal's algorithm, we have a set of edges that is part of an MST of G
- Proof of claim: Base case …
- Induction step:
  - Induction Assumption: before adding an edge we have a subgraph of an MST
  - We must show that after adding the next edge we have a subgraph of an MST
  - Details:

## Does Prim produce an MST?

- Proof similar to Kruskal (but slightly simpler)
- It's done in the textbook

## Recap: Prim's Algorithm for Minimal Spanning Tree

- Start with T as a single vertex of G (which *is* a MST for a single-node graph).
- for i = 1 to n – 1:
  - Among all edges of G that connect a vertex in T to a vertex that is not yet in T, add to T a minimum-weight edge.

At each stage, T is a MST for a connected subgraph of G

**We now examine Prim more closely**

# Main Data Structures for Prim

- Start with adjacency-list representation of G
- Let V be all of the vertices of G, and let $V_T$ the subset consisting of the vertices that we have placed in the tree so far
- We need a way to keep track of "fringe" edges
  - i.e. edges that have one vertex in $V_T$ and the other vertex in $V – V_T$
- Fringe edges need to be ordered by edge weight
  - E.g., in a priority queue
- What is the most efficient way to implement a priority queue?

# Prim detailed algorithm summary

- **Create a minheap** from the adjacency-list representation of G
  - Each heap entry contains a vertex and its weight
  - **The vertices in the heap are those not yet in T**
  - Weight associated with each vertex v is the minimum weight of an edge that connects v to some vertex in T
  - **If there is no such edge, v's weight is infinite**
    - **Initially all vertices except *start* are in heap, have infinite weight**
  - Vertices in the heap whose weights are not infinite are the fringe vertices
  - **Fringe vertices are candidates to be the next vertex (with its associated edge) added to the tree**
- **Loop:**
  - Delete min weight vertex from heap, add it to T
  - We may then be able to decrease the weights associated with one or vertices that are adjacent to v

# MinHeap overview

- We need an operation that a standard binary heap doesn't support:
  `decrease(vertex, newWeight)`
  – Decreases the value associated with a heap element
- Instead of putting vertices and associated edge weights directly in the heap:
  – Put them in an array called **key[]**
  – Put references to them in the heap

# Min Heap methods

| operation | description | run time |
|---|---|---|
| init(key) | build a MinHeap from the array of keys | Θ(n) |
| del() | delete and return (the location in key[ ] of ) the minimum element | Θ(log n) |
| isIn(w) | is vertex w currently in the heap? | Θ(1) |
| keyVal(w) | The weight associated with vertex w (minimum weight of an edge from that vertex to some adjacent vertex that is in the tree). | Θ(1) |
| decrease(w, newWeight) | changes the weight associated with vertex w to newWeight (which must be smaller than w's current weight) | Θ(log n) |