Minimum Cost Spanning Trees: Prim's Algorithm

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 5

Minimum cost spanning tree (MCST)

- Weighted undirected graph, $G = (V, E), W : E \to \mathbb{R}$
  - G assumed to be connected

Minimum cost spanning tree (MCST)

- Weighted undirected graph, $G = (V, E), W : E \to \mathbb{R}$
  - G assumed to be connected
- Find a minimum cost spanning tree
  - Tree connecting all vertices in V

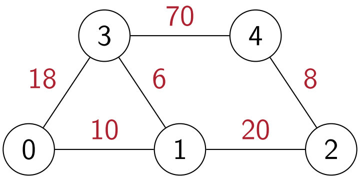
Minimum cost spanning tree (MCST)

- Weighted undirected graph, $G = (V, E), W : E \to \mathbb{R}$
  - G assumed to be connected
- Find a minimum cost spanning tree
  - Tree connecting all vertices in V
- Strategy
  - Incrementally grow the minimum cost spanning tree
  - Start with a smallest weight edge overall
  - Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

Minimum cost spanning tree (MCST)

- Weighted undirected graph, $G = (V, E), W : E \to \mathbb{R}$
  - G assumed to be connected
- Find a minimum cost spanning tree
  - Tree connecting all vertices in V
- Strategy
  - Incrementally grow the minimum cost spanning tree
  - Start with a smallest weight edge overall
  - Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

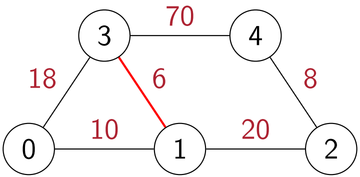
Example



Minimum cost spanning tree (MCST)

- Weighted undirected graph, $G = (V, E), W : E \to \mathbb{R}$
  - G assumed to be connected
- Find a minimum cost spanning tree
  - Tree connecting all vertices in V
- Strategy
  - Incrementally grow the minimum cost spanning tree
  - Start with a smallest weight edge overall
  - Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

Example

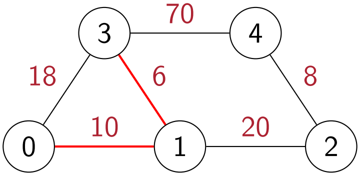


- Start with smallest edge, $(1, 3)$

Minimum cost spanning tree (MCST)

- Weighted undirected graph, $G = (V, E), W : E \to \mathbb{R}$
  - G assumed to be connected
- Find a minimum cost spanning tree
  - Tree connecting all vertices in V
- Strategy
  - Incrementally grow the minimum cost spanning tree
  - Start with a smallest weight edge overall
  - Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

Example

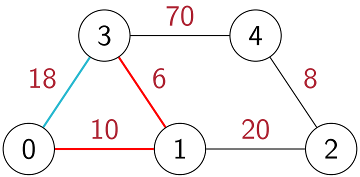


- Start with smallest edge, $(1, 3)$
- Extend the tree with $(1, 0)$

Minimum cost spanning tree (MCST)

- Weighted undirected graph, $G = (V, E), W : E \to \mathbb{R}$
  - G assumed to be connected
- Find a minimum cost spanning tree
  - Tree connecting all vertices in V
- Strategy
  - Incrementally grow the minimum cost spanning tree
  - Start with a smallest weight edge overall
  - Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

Example

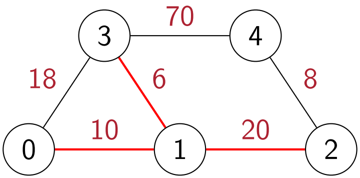


- Start with smallest edge, $(1,3)$
- Extend the tree with $(1,0)$
- Can't add $(0,3)$, forms a cycle

Minimum cost spanning tree (MCST)

- Weighted undirected graph, $G = (V, E), W : E \to \mathbb{R}$
  - G assumed to be connected
- Find a minimum cost spanning tree
  - Tree connecting all vertices in V
- Strategy
  - Incrementally grow the minimum cost spanning tree
  - Start with a smallest weight edge overall
  - Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

Example

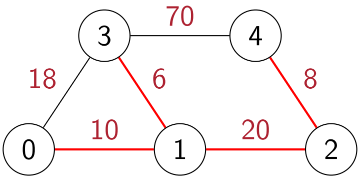


- Start with smallest edge, $(1, 3)$
- Extend the tree with $(1, 0)$
- Can't add $(0, 3)$, forms a cycle
- Instead, extend the tree with $(1, 2)$

Minimum cost spanning tree (MCST)

- Weighted undirected graph, $G = (V, E), W : E \to \mathbb{R}$
  - G assumed to be connected
- Find a minimum cost spanning tree
  - Tree connecting all vertices in V
- Strategy
  - Incrementally grow the minimum cost spanning tree
  - Start with a smallest weight edge overall
  - Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

Example



- Start with smallest edge, $(1, 3)$
- Extend the tree with $(1, 0)$
- Can't add $(0, 3)$, forms a cycle
- Instead, extend the tree with $(1, 2)$
- Extend the tree with $(2, 4)$

Prim's algorithm

- $G = (V, E)$, $W : E \to \mathbb{R}$

Prim's algorithm

- $G = (V, E)$, $W : E \to \mathbb{R}$
- Incrementally build an MCST
  - $TV \subseteq V$: tree vertices, already added to MCST
  - $TE \subseteq E$: tree edges, already added to MCST

Prim's algorithm

- $G = (V, E)$, $W : E \to \mathbb{R}$
- Incrementally build an MCST
  - $TV \subseteq V$: tree vertices, already added to MCST
  - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$

Prim's algorithm

- $G = (V, E)$, $W : E \to \mathbb{R}$
- Incrementally build an MCST
  - $TV \subseteq V$: tree vertices, already added to MCST
  - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
  - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST

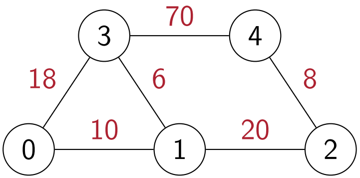
Prim's algorithm

- $G = (V, E)$, $W : E \to \mathbb{R}$
- Incrementally build an MCST
  - $TV \subseteq V$: tree vertices, already added to MCST
  - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
  - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST
- Repeat $n - 2$ times
  - Choose minimum weight edge $f = (u, v)$ such that $u \in TV$, $v \notin TV$
  - Add v to TV, f to TE

Prim's algorithm

- $G = (V, E)$, $W : E \to \mathbb{R}$
- Incrementally build an MCST
  - $TV \subseteq V$: tree vertices, already added to MCST
  - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
  - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST
- Repeat $n - 2$ times
  - Choose minimum weight edge $f = (u, v)$ such that $u \in TV$, $v \notin TV$
  - Add v to TV, f to TE

Example



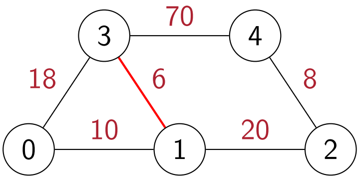
$TV = \emptyset$

$TE = \emptyset$

Prim's algorithm

- $G = (V, E)$, $W : E \to \mathbb{R}$
- Incrementally build an MCST
  - $TV \subseteq V$: tree vertices, already added to MCST
  - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
  - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST
- Repeat $n - 2$ times
  - Choose minimum weight edge $f = (u, v)$ such that $u \in TV$, $v \notin TV$
  - Add v to TV, f to TE

Example



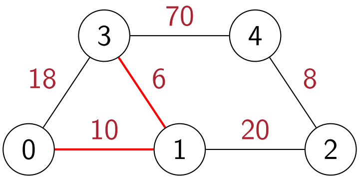
$TV = \{1, 3\}$

$TE = \{(1, 3)\}$

Prim's algorithm

- $G = (V, E)$, $W : E \to \mathbb{R}$
- Incrementally build an MCST
  - $TV \subseteq V$: tree vertices, already added to MCST
  - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
  - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST
- Repeat $n - 2$ times
  - Choose minimum weight edge $f = (u, v)$ such that $u \in TV$, $v \notin TV$
  - Add v to TV, f to TE

Example



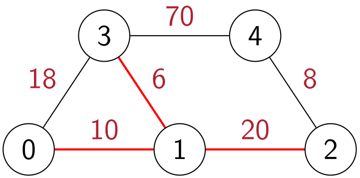
$TV = \{1, 3, 0\}$

$TE = \{(1, 3), (1, 0)\}$

Prim's algorithm

- $G = (V, E)$, $W : E \to \mathbb{R}$
- Incrementally build an MCST
  - $TV \subseteq V$: tree vertices, already added to MCST
  - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
  - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST
- Repeat $n - 2$ times
  - Choose minimum weight edge $f = (u, v)$ such that $u \in TV$, $v \notin TV$
  - Add v to TV, f to TE

Example



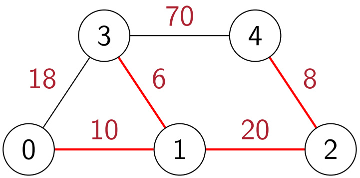
$TV = \{1, 3, 0, 2\}$

$TE = \{(1, 3), (1, 0), (1, 2)\}$

Prim's algorithm

- $G = (V, E)$, $W : E \to \mathbb{R}$
- Incrementally build an MCST
  - $TV \subseteq V$: tree vertices, already added to MCST
  - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
  - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST
- Repeat $n - 2$ times
  - Choose minimum weight edge $f = (u, v)$ such that $u \in TV$, $v \notin TV$
  - Add v to TV, f to TE

Example



$TV = \{1, 3, 0, 2, 4\}$

$TE = \{(1,3), (1,0), (1,2), (2,4)\}$

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

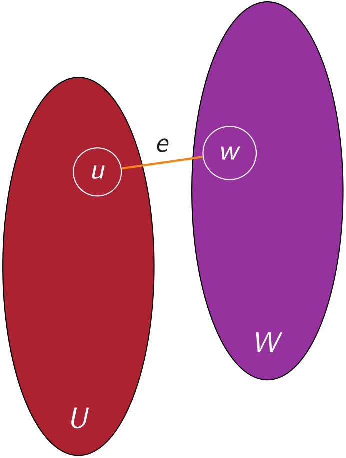
- Assume for now, all edge weights distinct

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- Assume for now, all edge weights distinct
- Let T be an MCST, $e \notin T$

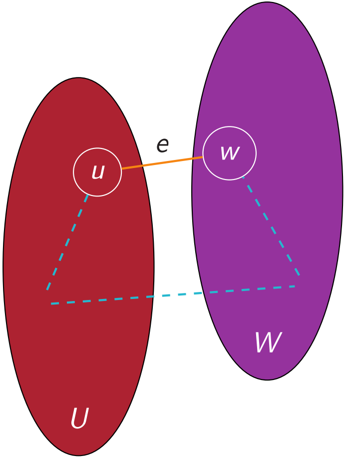


Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- Assume for now, all edge weights distinct
- Let T be an MCST, $e \notin T$
- T contains a path p from u to w

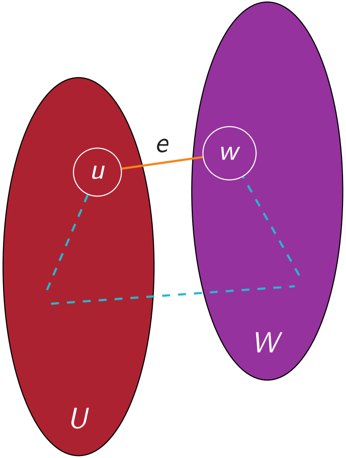


Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- Assume for now, all edge weights distinct
- Let T be an MCST, $e \notin T$
- T contains a path p from u to w
  - p starts in U, ends in W

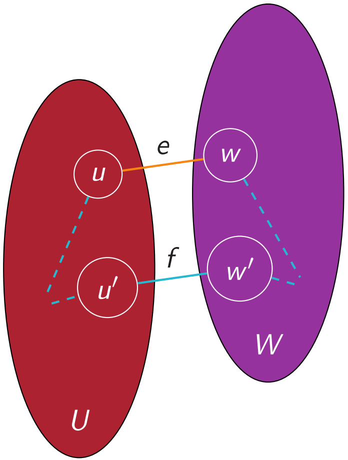


Correctness of Prim's algorithm

Minimum Separator Lemma

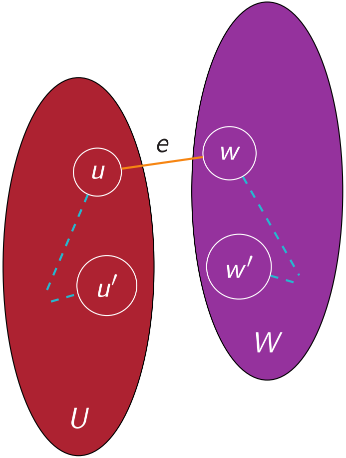
- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- Assume for now, all edge weights distinct
- Let T be an MCST, $e \notin T$
- T contains a path p from u to w
  - p starts in U, ends in W
  - Let $f = (u', w')$ be the first edge on p crossing from U to W



Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e



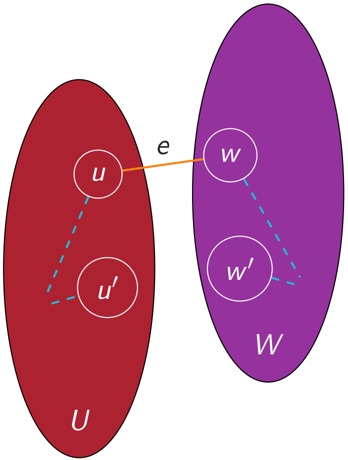
- Assume for now, all edge weights distinct
- Let T be an MCST, $e \notin T$
- T contains a path p from u to w
  - p starts in U, ends in W
  - Let $f = (u', w')$ be the first edge on p crossing from U to W
  - Drop f, add e to get a cheaper spanning tree

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- Assume for now, all edge weights distinct
- What if two edges have the same weight?

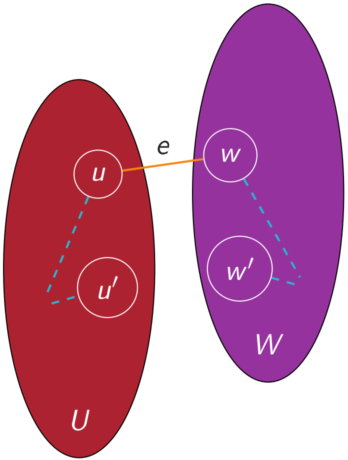


Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- Assume for now, all edge weights distinct
- What if two edges have the same weight?
- Assign each edge a unique index from 0 to $m - 1$

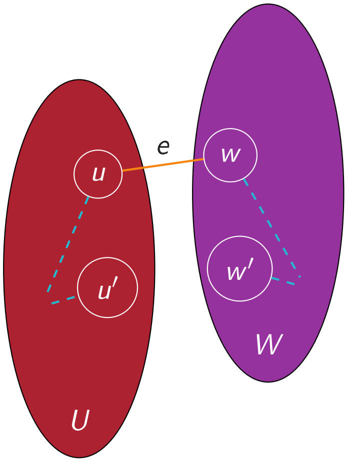


Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- Assume for now, all edge weights distinct
- What if two edges have the same weight?
- Assign each edge a unique index from 0 to $m - 1$
- Define $(e, i) < (f, j)$ if $W(e) < W(j)$ or $W(e) = W(j)$ and $i < j$



Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- In Prim's algorithm, TV and $W = V \setminus TV$ partition V

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- In Prim's algorithm, TV and $W = V \setminus TV$ partition V
- Algorithm picks smallest edge connecting TV and W, which must belong to every MCST

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- In Prim's algorithm, TV and $W = V \setminus TV$ partition V
- Algorithm picks smallest edge connecting TV and W, which must belong to every MCST

- In fact, for any $v \in V$, $\{v\}$ and $V \setminus \{v\}$ form a partition

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- In Prim's algorithm, TV and $W = V \setminus TV$ partition V
- Algorithm picks smallest edge connecting TV and W, which must belong to every MCST

- In fact, for any $v \in V$, $\{v\}$ and $V \setminus \{v\}$ form a partition
- The smallest weight edge leaving any vertex must belong to every MCST

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- In Prim's algorithm, TV and $W = V \setminus TV$ partition V
- Algorithm picks smallest edge connecting TV and W, which must belong to every MCST

- In fact, for any $v \in V$, $\{v\}$ and $V \setminus \{v\}$ form a partition
- The smallest weight edge leaving any vertex must belong to every MCST
- We started with overall minimum cost edge

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- In Prim's algorithm, TV and $W = V \setminus TV$ partition V
- Algorithm picks smallest edge connecting TV and W, which must belong to every MCST
- In fact, for any $v \in V$, $\{v\}$ and $V \setminus \{v\}$ form a partition
- The smallest weight edge leaving any vertex must belong to every MCST
- We started with overall minimum cost edge
- Instead, can start at any vertex v, with $TV = \{v\}$ and $TE = \emptyset$

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e

- In Prim's algorithm, TV and $W = V \setminus TV$ partition V
- Algorithm picks smallest edge connecting TV and W, which must belong to every MCST
- In fact, for any $v \in V$, $\{v\}$ and $V \setminus \{v\}$ form a partition
- The smallest weight edge leaving any vertex must belong to every MCST
- We started with overall minimum cost edge
- Instead, can start at any vertex v, with $TV = \{v\}$ and $TE = \emptyset$
- First iteration will pick minimum cost edge from v

Implementation

- Keep track of
  - `visited[v]` – is `v` in the spanning tree?
  - `distance[v]` – shortest distance from `v` to the tree
  - `TreeEdges` – edges in the current spanning tree

```
def primlist(WList):
  infinity = 1 + max([d for u in WList.keys()
                          for (v,d) in WList[u]])
  (visited,distance,TreeEdges) = ({},{},[])
  for v in WList.keys():
    (visited[v],distance[v]) = (False,infinity)
  visited[0] = True
  for (v,d) in WList[0]:
    distance[v] = d
  for i in WList.keys():
    (mindist,nextv) = (infinity,None)
    for u in WList.keys():
      for (v,d) in WList[u]:
        if visited[u] and (not visited[v]) and d < mindist:
          (mindist,nextv,nexte) = (d,v,(u,v))
    if nextv is None:
      break
    visited[nextv] = True
    TreeEdges.append(nexte)
    for (v,d) in WList[nextv]:
      if not visited[v]:
        distance[v] = min(distance[v],d)
  return(TreeEdges)
```

Implementation

- Keep track of
  - visited[v] – is v in the spanning tree?
  - distance[v] – shortest distance from v to the tree
  - TreeEdges – edges in the current spanning tree
- Initialize visited[v] to False, distance[v] to infinity

```
def primlist(WList):
  infinity = 1 + max([d for u in WList.keys()
                          for (v,d) in WList[u]])
  (visited,distance,TreeEdges) = ({},{},[])
  for v in WList.keys():
    (visited[v],distance[v]) = (False,infinity)
  visited[0] = True
  for (v,d) in WList[0]:
    distance[v] = d
  for i in WList.keys():
    (mindist,nextv) = (infinity,None)
    for u in WList.keys():
      for (v,d) in WList[u]:
        if visited[u] and (not visited[v]) and d < mindist:
          (mindist,nextv,nexte) = (d,v,(u,v))
    if nextv is None:
      break
    visited[nextv] = True
    TreeEdges.append(nexte)
    for (v,d) in WList[nextv]:
      if not visited[v]:
        distance[v] = min(distance[v],d)
  return(TreeEdges)
```

Implementation

- Keep track of
  - visited[v] – is v in the spanning tree?
  - distance[v] – shortest distance from v to the tree
  - TreeEdges – edges in the current spanning tree
- Initialize visited[v] to False, distance[v] to infinity
- First add vertex 0 to tree

```
def primlist(WList):
  infinity = 1 + max([d for u in WList.keys()
                             for (v,d) in WList[u]])
  (visited,distance,TreeEdges) = ({},{},[])
  for v in WList.keys():
    (visited[v],distance[v]) = (False,infinity)
  visited[0] = True
  for (v,d) in WList[0]:
    distance[v] = d
  for i in WList.keys():
    (mindist,nextv) = (infinity,None)
    for u in WList.keys():
      for (v,d) in WList[u]:
        if visited[u] and (not visited[v]) and d < mindist:
          (mindist,nextv,nexte) = (d,v,(u,v))
    if nextv is None:
      break
    visited[nextv] = True
    TreeEdges.append(nexte)
    for (v,d) in WList[nextv]:
      if not visited[v]:
        distance[v] = min(distance[v],d)
  return(TreeEdges)
```

Implementation

- Keep track of
  - visited[v] – is v in the spanning tree?
  - distance[v] – shortest distance from v to the tree
  - TreeEdges – edges in the current spanning tree
- Initialize visited[v] to False, distance[v] to infinity
- First add vertex 0 to tree
- Find edge (u,v) leaving the tree where distance[v] is minimum, add it to the tree, update distance[w] of neighbours

```
def primlist(WList):
  infinity = 1 + max([d for u in WList.keys()
                        for (v,d) in WList[u]])
  (visited,distance,TreeEdges) = ({},{},[])
  for v in WList.keys():
    (visited[v],distance[v]) = (False,infinity)
  visited[0] = True
  for (v,d) in WList[0]:
    distance[v] = d
  for i in WList.keys():
    (mindist,nextv) = (infinity,None)
    for u in WList.keys():
      for (v,d) in WList[u]:
        if visited[u] and (not visited[v]) and d < mindist:
          (mindist,nextv,nexte) = (d,v,(u,v))
    if nextv is None:
      break
    visited[nextv] = True
    TreeEdges.append(nexte)
    for (v,d) in WList[nextv]:
      if not visited[v]:
        distance[v] = min(distance[v],d)
  return(TreeEdges)
```

Complexity

- Initialization takes $(O(n)$

```
def primlist(WList):
  infinity = 1 + max([d for u in WList.keys()
                             for (v,d) in WList[u]])
  (visited,distance,TreeEdges) = ({},{},[])
  for v in WList.keys():
    (visited[v],distance[v]) = (False,infinity)
  visited[0] = True
  for (v,d) in WList[0]:
    distance[v] = d
  for i in WList.keys():
    (mindist,nextv) = (infinity,None)
    for u in WList.keys():
      for (v,d) in WList[u]:
        if visited[u] and (not visited[v]) and d < mindist:
          (mindist,nextv,nexte) = (d,v,(u,v))
    if nextv is None:
      break
    visited[nextv] = True
    TreeEdges.append(nexte)
    for (v,d) in WList[nextv]:
      if not visited[v]:
        distance[v] = min(distance[v],d)
  return(TreeEdges)
```

Complexity

- Initialization takes $(O(n)$
- Loop to add nodes to the tree runs $O(n)$ times

```
def primlist(WList):
  infinity = 1 + max([d for u in WList.keys()
                         for (v,d) in WList[u]])
  (visited,distance,TreeEdges) = ({},{},[])
  for v in WList.keys():
    (visited[v],distance[v]) = (False,infinity)
  visited[0] = True
  for (v,d) in WList[0]:
    distance[v] = d
  for i in WList.keys():
    (mindist,nextv) = (infinity,None)
    for u in WList.keys():
      for (v,d) in WList[u]:
        if visited[u] and (not visited[v]) and d < mindist:
          (mindist,nextv,nexte) = (d,v,(u,v))
    if nextv is None:
      break
    visited[nextv] = True
    TreeEdges.append(nexte)
    for (v,d) in WList[nextv]:
      if not visited[v]:
        distance[v] = min(distance[v],d)
  return(TreeEdges)
```

Complexity

- Initialization takes $(O(n)$
- Loop to add nodes to the tree runs $O(n)$ times
- Each iteration takes $O(m)$ time to find a node to add

```
def primlist(WList):
  infinity = 1 + max([d for u in WList.keys()
                        for (v,d) in WList[u]])
  (visited,distance,TreeEdges) = ({},{},[])
  for v in WList.keys():
    (visited[v],distance[v]) = (False,infinity)
  visited[0] = True
  for (v,d) in WList[0]:
    distance[v] = d
  for i in WList.keys():
    (mindist,nextv) = (infinity,None)
    for u in WList.keys():
      for (v,d) in WList[u]:
        if visited[u] and (not visited[v]) and d < mindist:
          (mindist,nextv,nexte) = (d,v,(u,v))
    if nextv is None:
      break
    visited[nextv] = True
    TreeEdges.append(nexte)
    for (v,d) in WList[nextv]:
      if not visited[v]:
        distance[v] = min(distance[v],d)
  return(TreeEdges)
```

Complexity

- Initialization takes $(O(n)$
- Loop to add nodes to the tree runs $O(n)$ times
- Each iteration takes $O(m)$ time to find a node to add
- Overall time is $O(mn)$, which could be $O(n^3)$!

```
def primlist(WList):
  infinity = 1 + max([d for u in WList.keys()
                         for (v,d) in WList[u]])
  (visited,distance,TreeEdges) = ({},{},[])
  for v in WList.keys():
    (visited[v],distance[v]) = (False,infinity)
  visited[0] = True
  for (v,d) in WList[0]:
    distance[v] = d
  for i in WList.keys():
    (mindist,nextv) = (infinity,None)
    for u in WList.keys():
      for (v,d) in WList[u]:
        if visited[u] and (not visited[v]) and d < mindist:
          (mindist,nextv,nexte) = (d,v,(u,v))
    if nextv is None:
      break
    visited[nextv] = True
    TreeEdges.append(nexte)
    for (v,d) in WList[nextv]:
      if not visited[v]:
        distance[v] = min(distance[v],d)
  return(TreeEdges)
```

Complexity

- Initialization takes $(O(n)$
- Loop to add nodes to the tree runs $O(n)$ times
- Each iteration takes $O(m)$ time to find a node to add
- Overall time is $O(mn)$, which could be $O(n^3)$!
- Can we do better?

```
def primlist(WList):
  infinity = 1 + max([d for u in WList.keys()
                         for (v,d) in WList[u]])
  (visited,distance,TreeEdges) = ({},{},[])
  for v in WList.keys():
    (visited[v],distance[v]) = (False,infinity)
  visited[0] = True
  for (v,d) in WList[0]:
    distance[v] = d
  for i in WList.keys():
    (mindist,nextv) = (infinity,None)
    for u in WList.keys():
      for (v,d) in WList[u]:
        if visited[u] and (not visited[v]) and d < mindist:
          (mindist,nextv,nexte) = (d,v,(u,v))
    if nextv is None:
      break
    visited[nextv] = True
    TreeEdges.append(nexte)
    for (v,d) in WList[nextv]:
      if not visited[v]:
        distance[v] = min(distance[v],d)
  return(TreeEdges)
```

Improved implementation

- For each `v`, keep track of its nearest neighbour in the tree
  - `visited[v]` – is `v` in the spanning tree?
  - `distance[v]` – shortest distance from `v` to the tree
  - `nbr[v]` – nearest neighbour of `v` in tree

```
def primlist2(WList):
  infinity = 1 + max([d for u in WList.keys()
                        for (v,d) in WList[u]])
  (visited,distance,nbr) = ({},{},{})
  for v in WList.keys():
    (visited[v],distance[v],nbr[v]) = (False,infinity,-1)
  visited[0] = True
  for (v,d) in WList[0]:
    (distance[v],nbr[v]) = (d,0)
  for i in range(1,len(WList.keys())):
    nextd = min([distance[v] for v in WList.keys()
                    if not visited[v]])
    nextvlist = [v for v in WList.keys()
                    if (not visited[v]) and
                       distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for (v,d) in WList[nextv]:
      if not visited[v]:
        (distance[v],nbr[v]) = (min(distance[v],d),nextv)
  return(nbr)
```

Improved implementation

- For each v, keep track of its nearest neighbour in the tree
  - visited[v] – is v in the spanning tree?
  - distance[v] – shortest distance from v to the tree
  - nbr[v] – nearest neighbour of v in tree
- Scan all non-tree vertices to find nextv with minimum distance

```
def primlist2(WList):
  infinity = 1 + max([d for u in WList.keys()
                        for (v,d) in WList[u]])
  (visited,distance,nbr) = ({},{},{})
  for v in WList.keys():
    (visited[v],distance[v],nbr[v]) = (False,infinity,-1)
  visited[0] = True
  for (v,d) in WList[0]:
    (distance[v],nbr[v]) = (d,0)
  for i in range(1,len(WList.keys())):
    nextd = min([distance[v] for v in WList.keys()
                    if not visited[v]])
    nextvlist = [v for v in WList.keys()
                    if (not visited[v]) and
                       distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for (v,d) in WList[nextv]:
      if not visited[v]:
        (distance[v],nbr[v]) = (min(distance[v],d),nextv)
  return(nbr)
```

Improved implementation

- For each v, keep track of its nearest neighbour in the tree
  - visited[v] – is v in the spanning tree?
  - distance[v] – shortest distance from v to the tree
  - nbr[v] – nearest neighbour of v in tree
- Scan all non-tree vertices to find nextv with minimum distance
- Then (nbr[nextv],nextv) is the tree edge to add

```
def primlist2(WList):
  infinity = 1 + max([d for u in WList.keys()
                        for (v,d) in WList[u]])
  (visited,distance,nbr) = ({},{},{})
  for v in WList.keys():
    (visited[v],distance[v],nbr[v]) = (False,infinity,-1)
  visited[0] = True
  for (v,d) in WList[0]:
    (distance[v],nbr[v]) = (d,0)
  for i in range(1,len(WList.keys())):
    nextd = min([distance[v] for v in WList.keys()
                   if not visited[v]])
    nextvlist = [v for v in WList.keys()
                   if (not visited[v]) and
                      distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for (v,d) in WList[nextv]:
      if not visited[v]:
        (distance[v],nbr[v]) = (min(distance[v],d),nextv)
  return(nbr)
```

Improved implementation

- For each `v`, keep track of its nearest neighbour in the tree
  - `visited[v]` – is `v` in the spanning tree?
  - `distance[v]` – shortest distance from `v` to the tree
  - `nbr[v]` – nearest neighbour of `v` in tree
- Scan all non-tree vertices to find `nextv` with minimum distance
- Then `(nbr[nextv],nextv)` is the tree edge to add
- Update `distance[v]` and `nbr[v]` for all neighbours of `nextv`

```
def primlist2(WList):
  infinity = 1 + max([d for u in WList.keys()
                        for (v,d) in WList[u]])
  (visited,distance,nbr) = ({},{},{})
  for v in WList.keys():
    (visited[v],distance[v],nbr[v]) = (False,infinity,-1)
  visited[0] = True
  for (v,d) in WList[0]:
    (distance[v],nbr[v]) = (d,0)
  for i in range(1,len(WList.keys())):
    nextd = min([distance[v] for v in WList.keys()
                    if not visited[v]])
    nextvlist = [v for v in WList.keys()
                    if (not visited[v]) and
                       distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for (v,d) in WList[nextv]:
      if not visited[v]:
        (distance[v],nbr[v]) = (min(distance[v],d),nextv)
  return(nbr)
```

Improved implementation — complexity

- Now the scan to find the next vertex to add is $O(n)$

```
def primlist2(WList):
  infinity = 1 + max([d for u in WList.keys()
                          for (v,d) in WList[u]])
  (visited,distance,nbr) = ({},{},{})
  for v in WList.keys():
    (visited[v],distance[v],nbr[v]) = (False,infinity,-1)
  visited[0] = True
  for (v,d) in WList[0]:
    (distance[v],nbr[v]) = (d,0)
  for i in range(1,len(WList.keys())):
    nextd = min([distance[v] for v in WList.keys()
                     if not visited[v]])
    nextvlist = [v for v in WList.keys()
                     if (not visited[v]) and
                        distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for (v,d) in WList[nextv]:
      if not visited[v]:
        (distance[v],nbr[v]) = (min(distance[v],d),nextv)
  return(nbr)
```

Improved implementation — complexity

- Now the scan to find the next vertex to add is $O(n)$
- Very similar to Dijkstra's algorithm, except for the update rule for distance

```
def primlist2(WList):
  infinity = 1 + max([d for u in WList.keys()
                        for (v,d) in WList[u]])
  (visited,distance,nbr) = ({},{},{})
  for v in WList.keys():
    (visited[v],distance[v],nbr[v]) = (False,infinity,-1)
  visited[0] = True
  for (v,d) in WList[0]:
    (distance[v],nbr[v]) = (d,0)
  for i in range(1,len(WList.keys())):
    nextd = min([distance[v] for v in WList.keys()
                    if not visited[v]])
    nextvlist = [v for v in WList.keys()
                    if (not visited[v]) and
                        distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for (v,d) in WList[nextv]:
      if not visited[v]:
        (distance[v],nbr[v]) = (min(distance[v],d),nextv)
  return(nbr)
```

Improved implementation — complexity

- Now the scan to find the next vertex to add is $O(n)$
- Very similar to Dijkstra's algorithm, except for the update rule for distance
- Like Dijkstra's algorithm, this is still $O(n^2)$ even for adjacency lists

```
def primlist2(WList):
  infinity = 1 + max([d for u in WList.keys()
                          for (v,d) in WList[u]])
  (visited,distance,nbr) = ({},{},{})
  for v in WList.keys():
    (visited[v],distance[v],nbr[v]) = (False,infinity,-1)
  visited[0] = True
  for (v,d) in WList[0]:
    (distance[v],nbr[v]) = (d,0)
  for i in range(1,len(WList.keys())):
    nextd = min([distance[v] for v in WList.keys()
                     if not visited[v]])
    nextvlist = [v for v in WList.keys()
                     if (not visited[v]) and
                        distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for (v,d) in WList[nextv]:
      if not visited[v]:
        (distance[v],nbr[v]) = (min(distance[v],d),nextv)
  return(nbr)
```

Improved implementation — complexity

- Now the scan to find the next vertex to add is $O(n)$
- Very similar to Dijkstra's algorithm, except for the update rule for distance
- Like Dijkstra's algorithm, this is still $O(n^2)$ even for adjacency lists
- With a more clever data structure to extract the minimum, we can do better

```
def primlist2(WList):
  infinity = 1 + max([d for u in WList.keys()
                         for (v,d) in WList[u]])
  (visited,distance,nbr) = ({},{},{})
  for v in WList.keys():
    (visited[v],distance[v],nbr[v]) = (False,infinity,-1)
  visited[0] = True
  for (v,d) in WList[0]:
    (distance[v],nbr[v]) = (d,0)
  for i in range(1,len(WList.keys())):
    nextd = min([distance[v] for v in WList.keys()
                     if not visited[v]])
    nextvlist = [v for v in WList.keys()
                     if (not visited[v]) and
                        distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for (v,d) in WList[nextv]:
      if not visited[v]:
        (distance[v],nbr[v]) = (min(distance[v],d),nextv)
  return(nbr)
```

Summary

- Prim's algorithm grows an MCST starting with any vertex
- At each step, connect one more vertex to the tree using minimum cost edge from inside the tree to outside the tree
- Correctness follows from Minimum Separator Lemma
- Implementation similar to Dijkstra's algorithms
  - Update rule for distance is different
- Complexity is $O(n^2)$
  - Even with adjacency lists
  - Bottleneck is identifying unvisited vertex with minimum distance
  - Need a better data structure to identify and remove minimum (or maximum) from a collection