14.7.1 Prim-Jarník Algorithm

In the Prim-Jarník algorithm, we grow a minimum spanning tree from a single cluster starting from some "root" vertex s. The main idea is similar to that of Dijkstra's algorithm. We begin with some vertex s, defining the initial "cloud" of vertices C. Then, in each iteration, we choose a minimum-weight edge $e = (u,v)$, connecting a vertex u in the cloud C to a vertex v outside of C. The vertex v is then brought into the cloud C and the process is repeated until a spanning tree is formed. Again, the crucial fact about minimum spanning trees comes into play, for by always choosing the smallest-weight edge joining a vertex inside C to one outside C, we are assured of always adding a valid edge to the MST.

To efficiently implement this approach, we can take another cue from Dijkstra's algorithm. We maintain a label $D[v]$ for each vertex v outside the cloud C, so that $D[v]$ stores the weight of the minimum observed edge for joining v to the cloud C. (In Dijkstra's algorithm, this label measured the full path length from starting vertex s to v, including an edge (u,v).) These labels serve as keys in a priority queue used to decide which vertex is next in line to join the cloud. We give the pseudo-code in Code Fragment 14.15.

Algorithm PrimJarnik(G):
  Input: An undirected, weighted, connected graph G with n vertices and m edges
  Output: A minimum spanning tree T for G

 Pick any vertex s of G
 $D[s] = 0$
 for each vertex $v \neq s$ **do**
  $D[v] = \infty$
 Initialize $T = \emptyset$.
 Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
 while Q is not empty **do**
  (u,e) = value returned by Q.remove_min()
  Connect vertex u to T using edge e.
  for each edge $e' = (u,v)$ such that v is in Q **do**
   {check if edge (u,v) better connects v to T}
   if $w(u,v) < D[v]$ **then**
    $D[v] = w(u,v)$
    Change the key of vertex v in Q to $D[v]$.
    Change the value of vertex v in Q to (v,e').
 return the tree T

Code Fragment 14.15: The Prim-Jarník algorithm for the MST problem.

Analyzing the Prim-Jarník Algorithm

The implementation issues for the Prim-Jarník algorithm are similar to those for Dijkstra's algorithm, relying on an adaptable priority queue Q (Section 9.5.1). We initially perform n insertions into Q, later perform n extract-min operations, and may update a total of m priorities as part of the algorithm. Those steps are the primary contributions to the overall running time. With a heap-based priority queue, each operation runs in $O(\log n)$ time, and the overall time for the algorithm is $O((n+m)\log n)$, which is $O(m \log n)$ for a connected graph. Alternatively, we can achieve $O(n^2)$ running time by using an unsorted list as a priority queue.

Illustrating the Prim-Jarník Algorithm

We illustrate the Prim-Jarník algorithm in Figures 14.20 through 14.21.

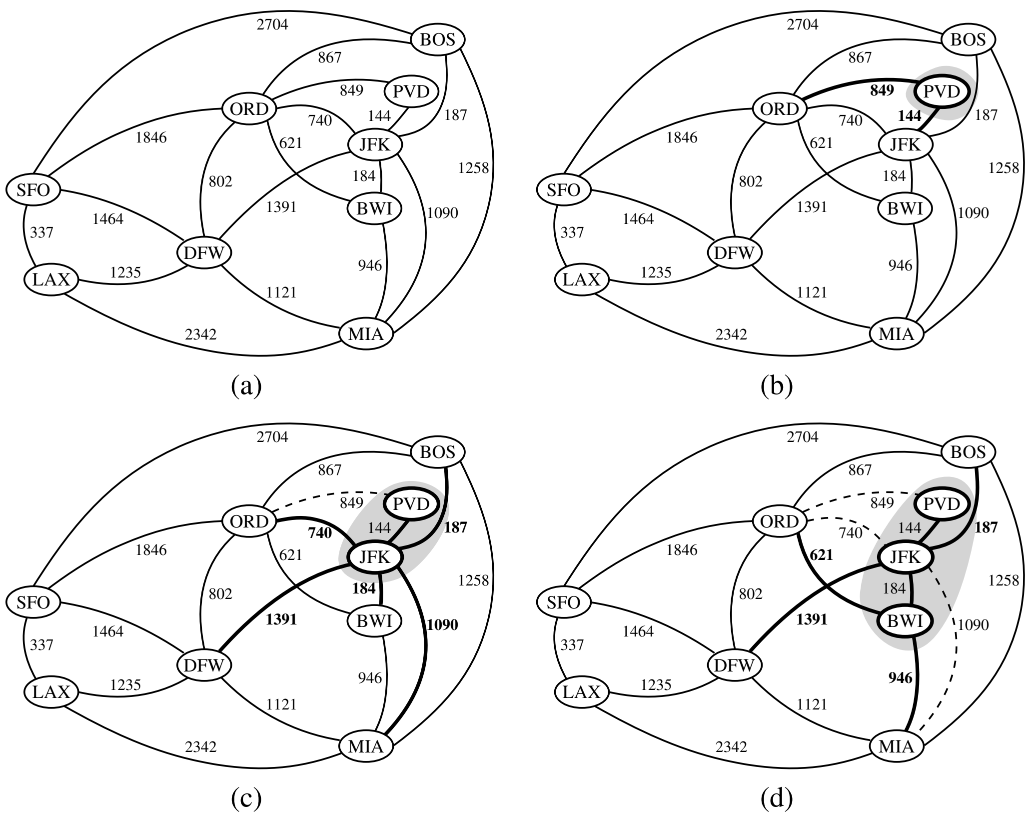

Figure 14.20: An illustration of the Prim-Jarník MST algorithm, starting with vertex PVD. (Continues in Figure 14.21.)

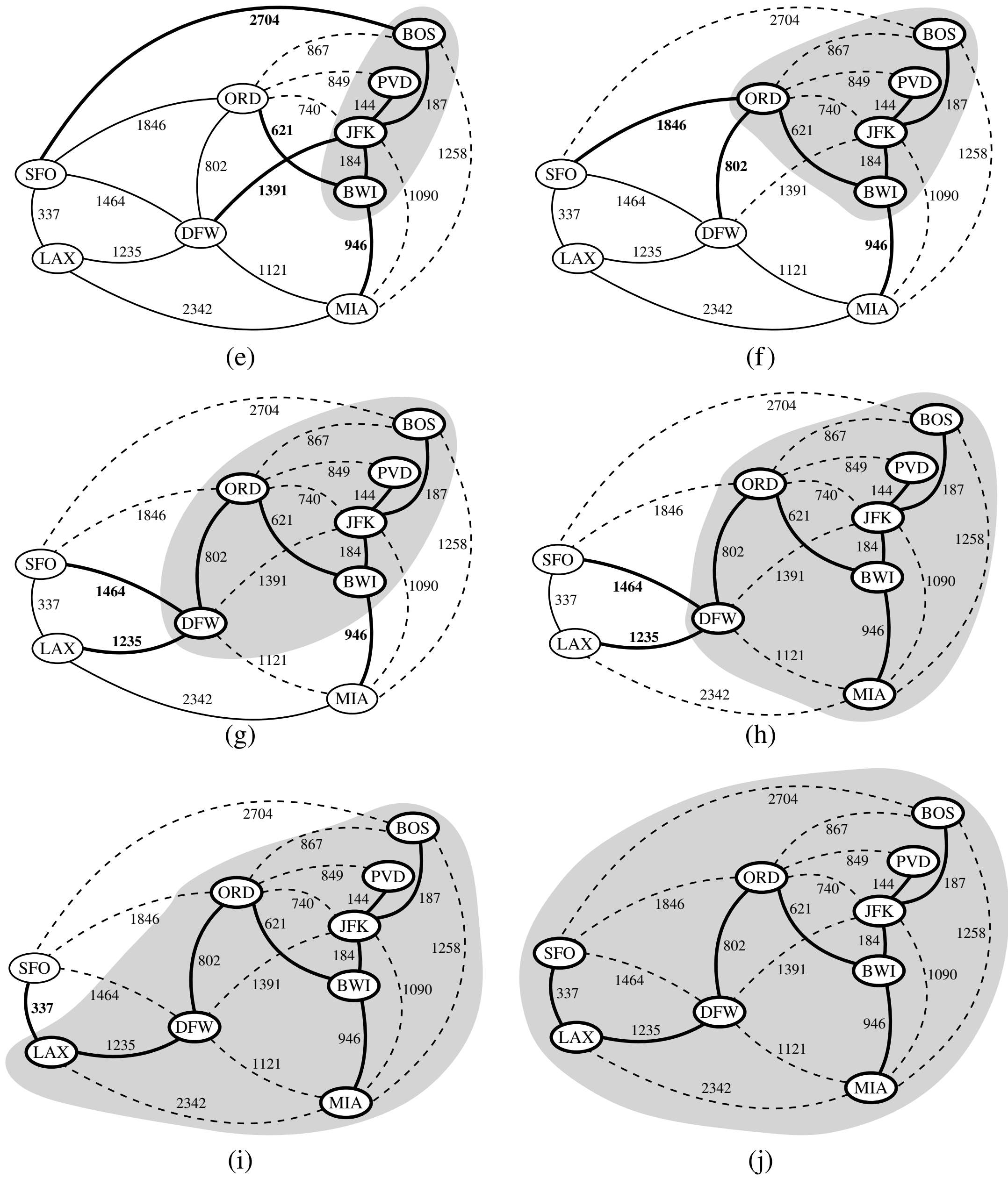

Figure 14.21: An illustration of the Prim-Jarník MST algorithm. (Continued from Figure 14.20.)

Python Implementation

Code Fragment 14.16 presents a Python implementation of the Prim-Jarník algorithm. The MST is returned as an unordered list of edges.

```
def MST_PrimJarnik(g):
  """Compute a minimum spanning tree of weighted graph g.

  Return a list of edges that comprise the MST (in arbitrary order).
  """
  d = { }                               # d[v] is bound on distance to tree
  tree = [ ]                            # list of edges in spanning tree
  pq = AdaptableHeapPriorityQueue( )    # d[v] maps to value (v, e=(u,v))
  pqlocator = { }                       # map from vertex to its pq locator

  # for each vertex v of the graph, add an entry to the priority queue, with
  # the source having distance 0 and all others having infinite distance
  for v in g.vertices( ):
    if len(d) == 0:                               # this is the first node
      d[v] = 0                                    # make it the root
    else:
      d[v] = float('inf')                         # positive infinity
    pqlocator[v] = pq.add(d[v], (v,None))

  while not pq.is_empty( ):
    key,value = pq.remove_min( )
    u,edge = value                                # unpack tuple from pq
    del pqlocator[u]                              # u is no longer in pq
    if edge is not None:
      tree.append(edge)                           # add edge to tree
    for link in g.incident_edges(u):
      v = link.opposite(u)
      if v in pqlocator:                          # thus v not yet in tree
        # see if edge (u,v) better connects v to the growing tree
        wgt = link.element( )
        if wgt < d[v]:                            # better edge to v?
          d[v] = wgt                              # update the distance
          pq.update(pqlocator[v], d[v], (v, link))  # update the pq entry
  return tree
```

Code Fragment 14.16: Python implementation of the Prim-Jarník algorithm for the minimum spanning tree problem.