CIS 5511. Programming Techniques

Graphs (2)

1. Minimum spanning trees

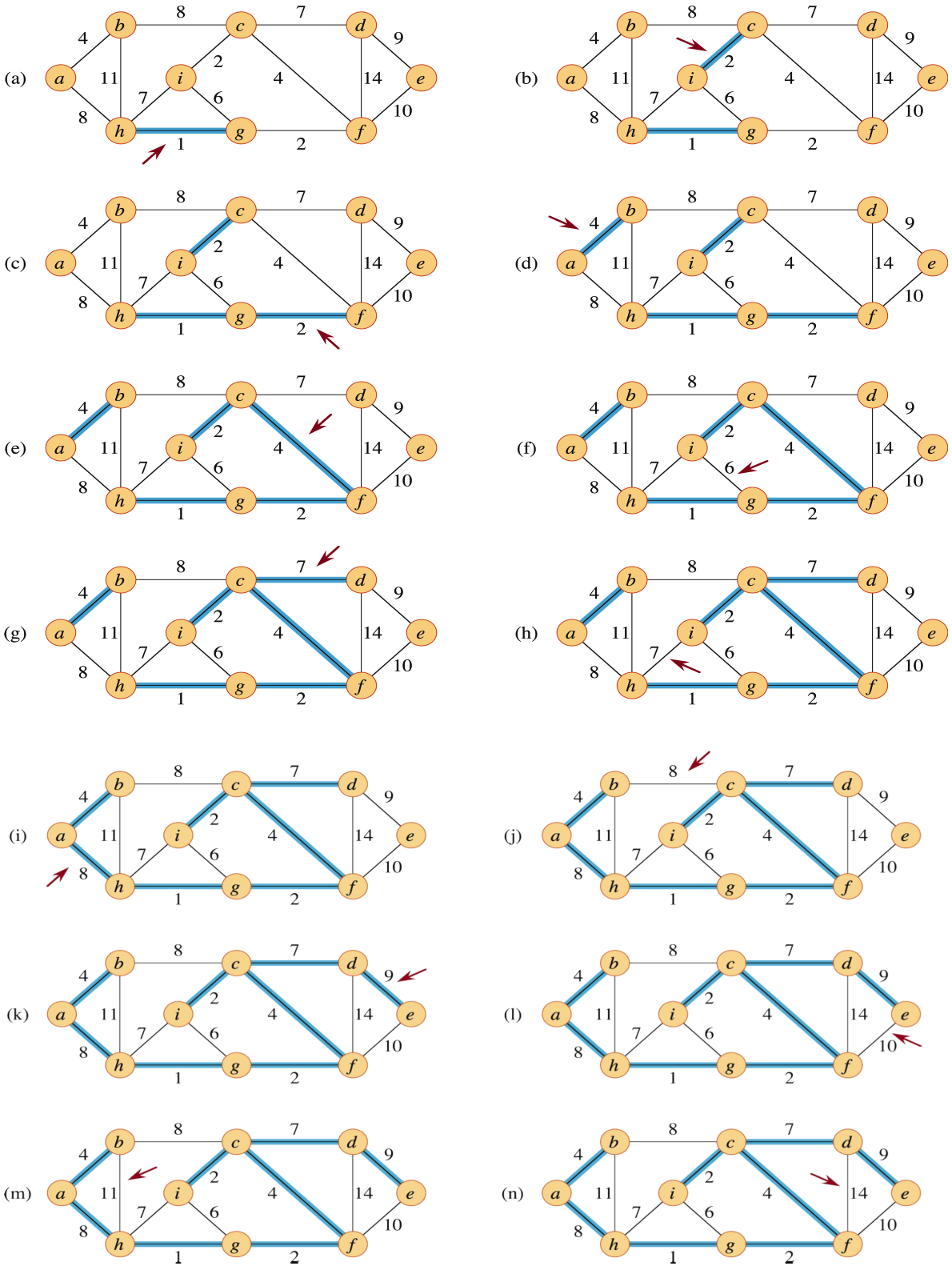
For an undirected and connected graph G = (V, E), its *spanning tree* is a subgraph G' = (V, E'), which is still connected and $|E'| = |V| - 1$, so G' does not have any cycle. This type of *tree* is different from the ones introduced previously, as there is no root specified.

If G is weighted, its *minimum spanning tree* has a minimum value of total weight $\sum w(u, v)$ [for $(u, v) \in E'$] among all spanning trees of G.

Kruskal's algorithm each time adds an edge with the least *w*, and connects two previously unconnected subgraphs. The algorithm uses a set to represent the vertices in a subtree, and Find-Set(*u*) identifies the set in which vertex *u* belongs.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  create a single list of the edges in G.E
5  sort the list of edges into monotonically increasing order by weight w
6  for each edge (u, v) taken from the sorted list in order
7      if FIND-SET(u) ≠ FIND-SET(v)
8          A = A ∪ {(u, v)}
9          UNION(u, v)
10 return A
```

Example:

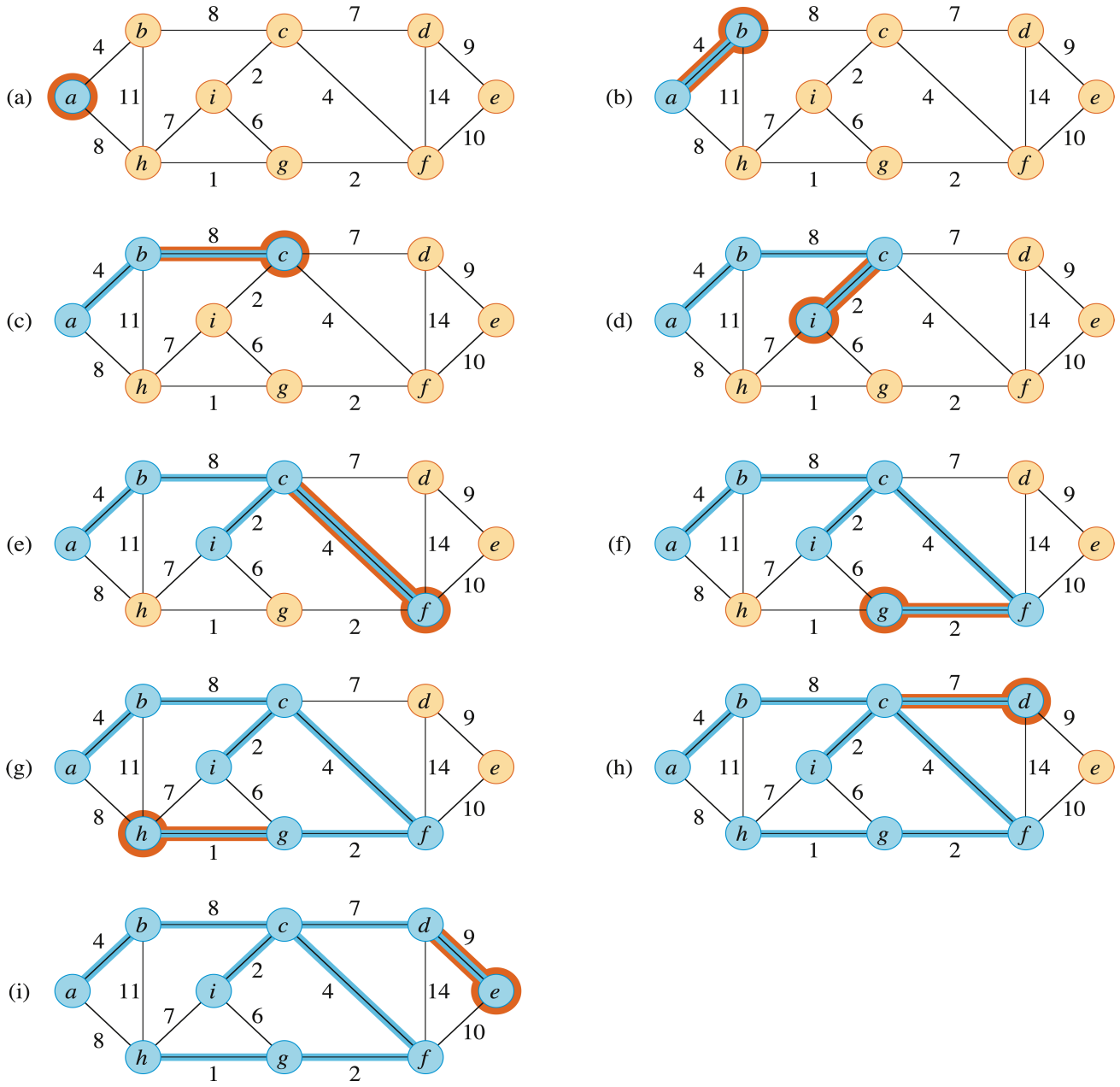


Kruskal's algorithm is a greedy algorithm, and takes O(E lg E) time, which is the same as O(E lg V).

Prim's algorithm adds vertices into the tree one by one, according to their distance to the tree built so far, starting from a root *r*. *Q* is a priority queue of vertices, where for each vertex *v*, key[*v*] is the minimum weight of any edge connecting *v* to a vertex in the tree.

```
MST-PRIM(G, w, r)
 1  for each vertex u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = ∅
 6  for each vertex u ∈ G.V
 7      INSERT(Q, u)
 8  while Q ≠ ∅
 9      u = EXTRACT-MIN(Q)          // add u to the tree
10      for each vertex v in G.Adj[u]   // update keys of u's non-tree neighbors
11          if v ∈ Q and w(u, v) < v.key
12              v.π = u
13              v.key = w(u, v)
14              DECREASE-KEY(Q, v, w(u, v))
```

Example:



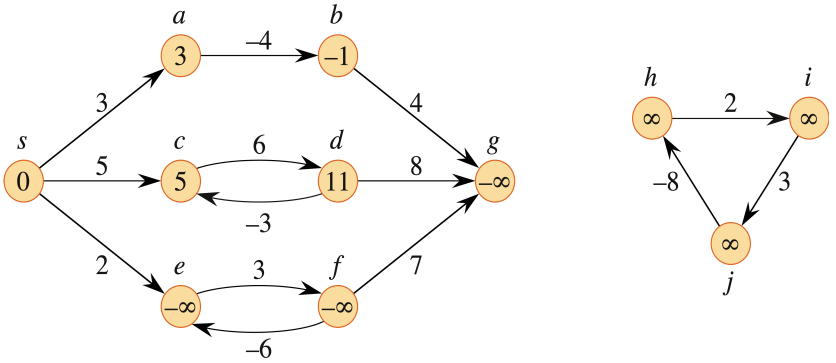
Prim's algorithm also takes O(E lg V) time.

2. Single-source shortest paths

In a weighted graph, one type of optimization problem is to find the shortest path between vertices, and it can be one-to-one, one-to-many, or many-to-many. In particular, "single-source" means to find the shortest paths from one source vertex to all the other vertices, so is the one-to-many case. Here "distance" or "weight" can represent many different measurements, so can have any finite (positive, negative, or zero) value.

The weight of a path is the sum of the weights of the edges in the path. A shortest path is a path whose weight has the lowest value among all paths with the same source and destination. A section of a shortest path is also a shortest path.

In general, a shortest path can be defined in graphs that are either directed or undirected, with or without cycle. However, a shortest path cannot contain cycle. Therefore, in graph G = (V, E), the shortest path contains less than |V| edges. For example, in the following graph the lengths of the shorted paths from *s* to each vertex is marked, which include positive and negative infinite as special cases.



A trivial solution to the problem is to traverse the graph and enumerate all possible paths, and compare their distances. However, this solution is usually too time-consuming to be useful.

If all edges are equally-weighted, the breadth-first search algorithm BFS introduced previously finds shortest paths from a source vertex to all other vertices. However, it does not work if the edges have different weights.

To represent the shortest paths, an array *d* is used to record the shortest distance found so far from the source to each vertex, and an array π for the predecessor of each vertex on that path. The following algorithm initializes the data structure.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

Many shortest-path algorithms use the "relaxation" step, which maintains the shortest paths from a given source to *v* found so far, and try to improve them when a new vertex *u* is taken into consideration. The matrix *w* stores the weights of the edges.

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
3        v.π = u
```

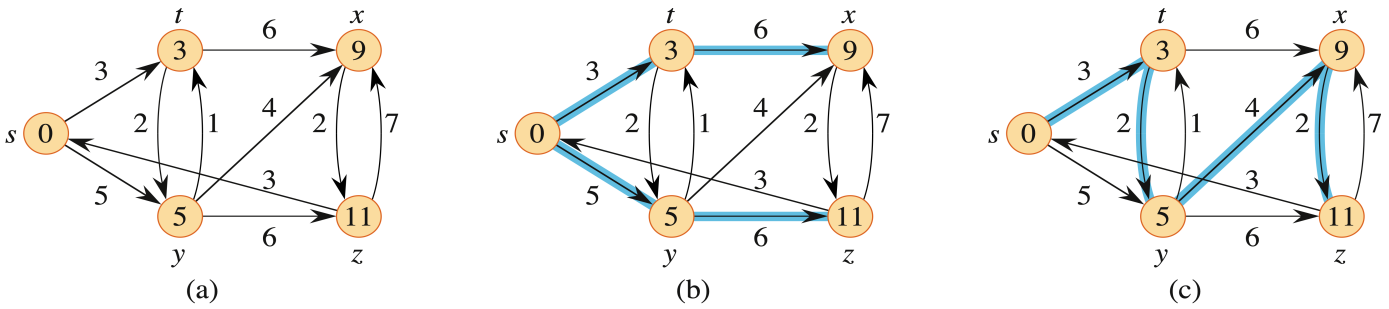
There are various ways to use this step to get the shortest paths, though it is important to see that after the distance to u is undated, the same step may need to be repeated.

Bellman-Ford algorithm processes the graph $|V| - 1$ passes. In each pass, the edges are tried one-by-one to relax the distance. After that, if there is still a possible relaxation, the graph must contain negatively-weighted cycle, so there is no shortest path to the involved vertices.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3        for each edge (u, v) ∈ G.E
4              RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6        if v.d > u.d + w(u, v)
7              return FALSE
8  return TRUE
```

The running time is O(V E).

Example:



If the graph is a DAG, there are faster solutions. The following algorithm topologically sorts the vertices first, then determine the shortest paths for each vertex in that order. It runs in $\Theta(V + E)$ time, which comes from topological sorting (and DFS).

Example:

Dijkstra's algorithm works for weighted graphs without any negative weight. It repeatedly selects the vertex with the shorted path, and uses it to relax the paths to other vertices.

This is a greedy algorithm. Its running time is O((V + E) lg V).

Example:

If only the shortest path to a single destination is needed, stop the algorithm at that vertex. The worst-case complexity will be the same.

3. All-pairs shortest paths

If we want the shortest paths between every pair of vertices, it is inefficient to repeat an algorithm for single-source with each vertex as source.

Many algorithms extend the weight matrix W to L that records the shortest paths found so far, plus a predecessor matrix Π, where π_{ij} is the predecessor of j on some shortest path from i to j, i.e., the last stop. Given it, the following algorithm prints the shortest path.

The following dynamic-programming algorithm extends shortest paths starting with single edges, and in each step tries to add one more edge. Here $n = |V|$.

Each call to the above algorithm will extend the length of path under consideration by one edge. Therefore, if we start with L = W, and repeatedly call the algorithm $n - 1$ times, we will get a matrix for the shortest paths. This algorithm takes $\Theta(n^4)$ time:

Example:

The above algorithm can be improved by updating the loop variable not as m = m + 1 but as m = 2m, and change the W in line 5 to $L^{(m-1)}$, so as to achieve $\Theta(n^3 \lg n)$ time.

Floyd-Warshall algorithm solves the problem by adding one possible intermediate vertex into the shortest paths in each step:

For the same problem, the intermediate results are:

The running time of the above algorithm is $\Theta(n^3)$, so is more efficient than the above two.

A similar algorithm calculates the transitive closure of a graph, where $T^{(n)}_{ij} = 1$ if and only if there is a path from i to j.