Algorithms and Data Structures

Charles A. Wuethrich

Bauhaus-University Weimar - CogVis/MMC

May 8, 2019

Stack or Queue-based algorithms / Recursive algorithms

- Introduction
- Overview
- Max span in a graph
- Breadth search
- Weighted Graphs
- Minimal spanning tree
- Shortest path
- Flux in a network
- Pairing
- Stable marriage

What is Stack/Queue-based?

- Remember what a Queue is?
  - FIFO
- Remember what a Stack is?
  - FILO
- What do we need them for?
- Collection items to work with later on.
- Collection items to work with later on.

Whats the difference?

- What is a queue good for?
  - Breath search
- What is a stack good for?
  - Depth search

The algorithms we talk about!

- Graphs
  - Maximum span
- Depth and breath search
  - Recursive on stack
  - Non-recursive on matrix
- Applications
  - Connected components
  - Depth search in a connected Graph
- Minimum spanning tree
- Priority search
- Applications:
  - Shortest paths
  - Prim's algo for dense graphs
- Find shortest lines joining n points
- Flux in a network
- Pairing

Depth search in a Graph

- Remember the possible representations
  - Adjacency list
  - Adjacency matrix

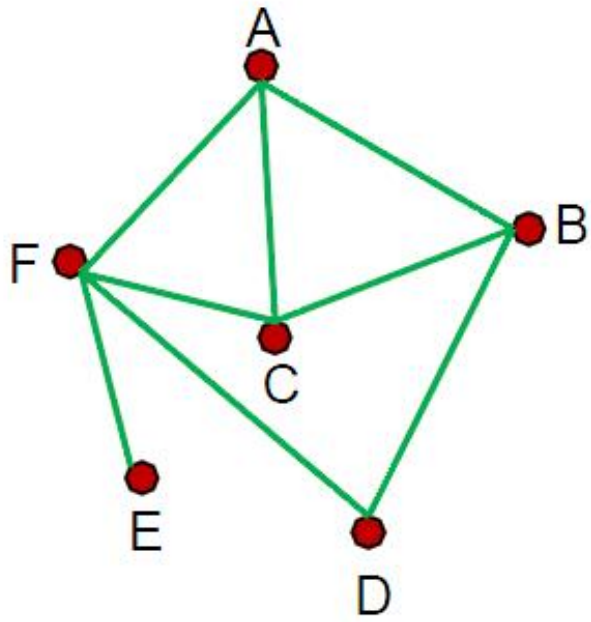

Figure: Graph.

Depth search in a Graph

- Remember the possible representations
  - Adjacency list
  - Adjacency matrix

A	**F**	**B**	**C**	
B	A	C	D	
C	A	B	F	
D	B	F		
E	F			
F	A	C	E	D

Figure: Adjacency list.

Depth search in a Graph

- Remember the possible representations
  - Adjacency list
  - Adjacency matrix

	A	B	C	D	E	F
A	1	1	1	0	0	1
B	1	1	1	1	0	0
C	1	1	1	0	0	1
D	0	1	0	1	0	1
E	0	0	0	0	1	1
F	1	0	1	1	1	1

Figure: Adjacency matrix.

Max span / depth of a graph

- Here a simple method to compute the max span of a graph from a starting point
  - Use list representation
  - Set all knots with initial variable v=0
  - Move from one starting knot
  - Look up in its adjacency list, increment counter by 1, and visit neighbouring nodes by marking them with counter: if adjacent node already marked, leave it alone, else keep marking on until no new knots can be visited

Max span / depth of a graph

- Notes:
  - This algorithm tell us if a graph is connected
  - Also connected components are computed: (all non connected knots have variable=0)
  - And also cycles can be computed: (when a node is visited twice)
- Execution time: O(V + E)
- Same algo can be used on Adjacency matrix representation, but here complexity= V2
- This due to the fact that lines in matrix have to be searched through for neighbours

Max span / depth of a graph

- Note that this implementation is recursive
- Note also that the method used can be seen as a tree traversal algorithm
- There is a simple way of unrolling recursion just as has been done for tree traversal

Max span / depth of a graph

```
void  search()
{
 int  k;
 for(k=1;k<=v; k++)  val [k]=unseen;
 for(k=1;k<=v; k++)
  if(val [k]= unseen) visit(k);
}
void visit(int k)//depth search Adjacency matrix
 {
 int t;
 val[k]=++id;
 for(t=1;t<=v;t++)
  if (a[k][t]=!0)
  if (a[k][t]==unseen)
   visit(t)
 }
```

non-recursive Max span / depth of a graph

- The idea is to use a stack for storing unexplored neighbours
- Nodes that have been not been completely processed are deposited in a stack
- Visiting a node means:
  - search its neighbors
  - deposit in the stack all nodes which are not processed or already in the stack

non-recursive Max span / depth of a graph

Idea:

- The implementation will foresee per node one variable indicating its status:
  - 0: non visited
  - $>$0: visited
  - -1: in the stack
- Note that the nodes are not traversed in the same way as the recursive algorithm (nodes: inverted when deposited in stack)
- The algorithm propagates very fast in depth
- Note that this algorithm can be used to find connected components of a graph

non-recursive Max span / depth of a graph

```
stack stack(max v);
void visit(int k)//nonrecursive depth search Adjacency list
{
 struct node *t;
 stack.push(k);
 while(!stack.empty())
 {
  k=stac.pop(); val[k]=++id;
  for(t=adj[k]; t!=z; t=t->next)
   if(val[t->v]==unseen)
   {
    stack.push(t->v);val[t->v]=-1;
   }
 }
}
```

Breadth search

- Similar as tree traversing, a FIFO queue can be used to save the nodes.
- Implementation exactly same, except for usage of a queue
- Resulting algorithm searches in order of distance from start node
- Easier to implement as non-recursive, while depth search easier for recursive implementation

Breadth search

```
Queue queue(max v);
void visit(int k)//Breadth search Adjacency list
{
 struct node *t;
 queue.put(k);
 while(!queue.empty())
 {
  k=queue.get(); val[k]=++id;
  for(t=adj[k]; t!=z; t=t->next)
   if(val[t->v]==unseen)
   {
    queue.put(t->v);val[t->v]=-1;
   }
 }
}
```

Applications of breadth search

- Very old use of depth search is in labyrinths
- Here one must save path followed to be able to track back path
- Easy to backtrack: search is always done among neighboring nodes.

Recap weighted graphs

- A weighted graph is a graph such that each edge has a weight (cost) associated to it
- Most common use: cost minimization
- Two problems are the most common:
  - Find minimal cost path covering all nodes *Minimal spanning tree, traveling salesman*
  - Find cheapest path between two nodes *Shortest path*

Prim-Jarnik: Minimal spanning tree

- Minimal spanning tree of a weighted graph: set of edges so that the sum of edge weights $<=$ all other spanning trees
- Note that its not unique.
- Also spanning trees are not unique: how do I know it's the right one?

Prim-Jarnik: Minimal spanning tree

- There is a property which helps: For any partition of a graph, the minimal spanning tree contains the shortest edges joining one partition with the other one
- Minimal spanning tree construction: Take one node, add to it the 'nearest' node
  - i.e. add edge with smallest weight adjacent to any up to date visited nodes
- Note that this adding is done on all nodes visited
- Property guarantees new node belongs to spanning tree

Prim-Jarnik: Minimal spanning tree

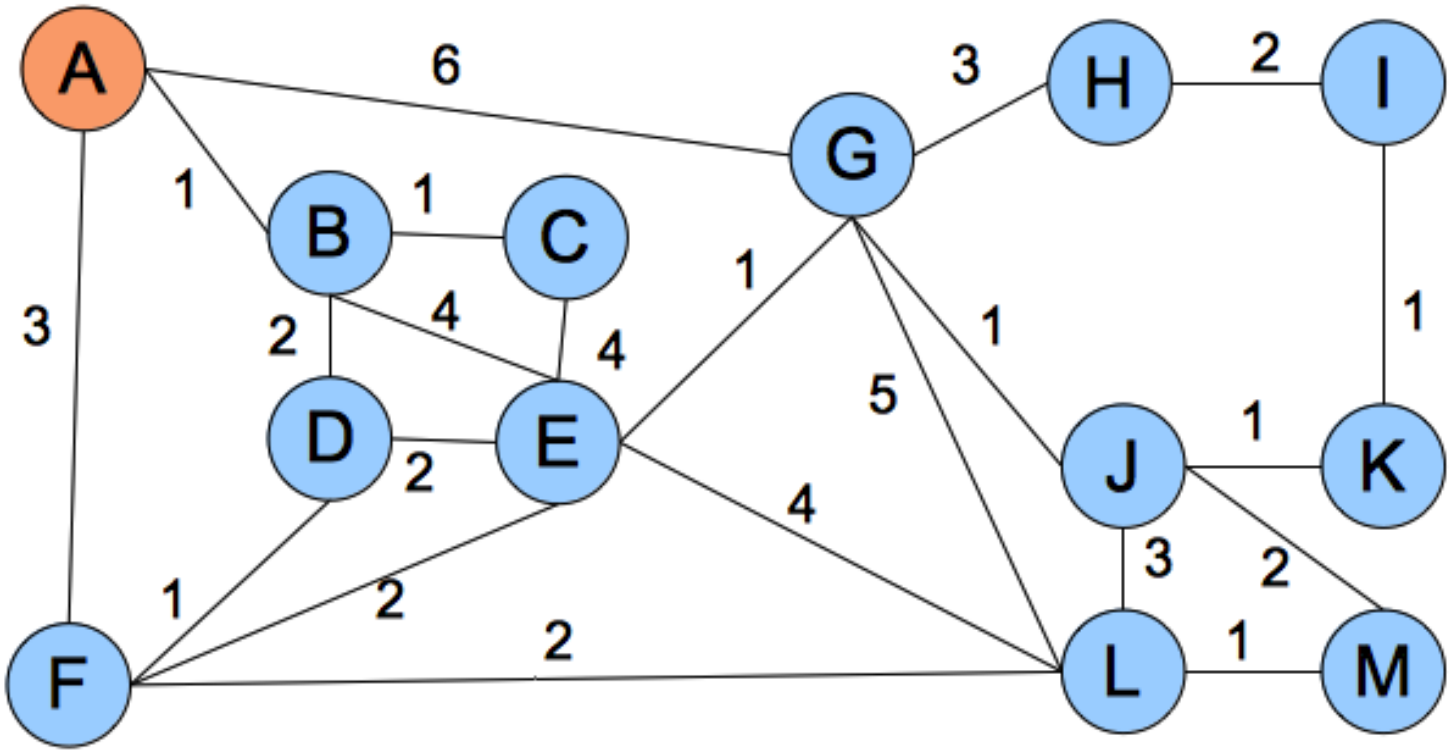

Figure: Spanning tree construction.

Prim-Jarnik: Minimal spanning tree

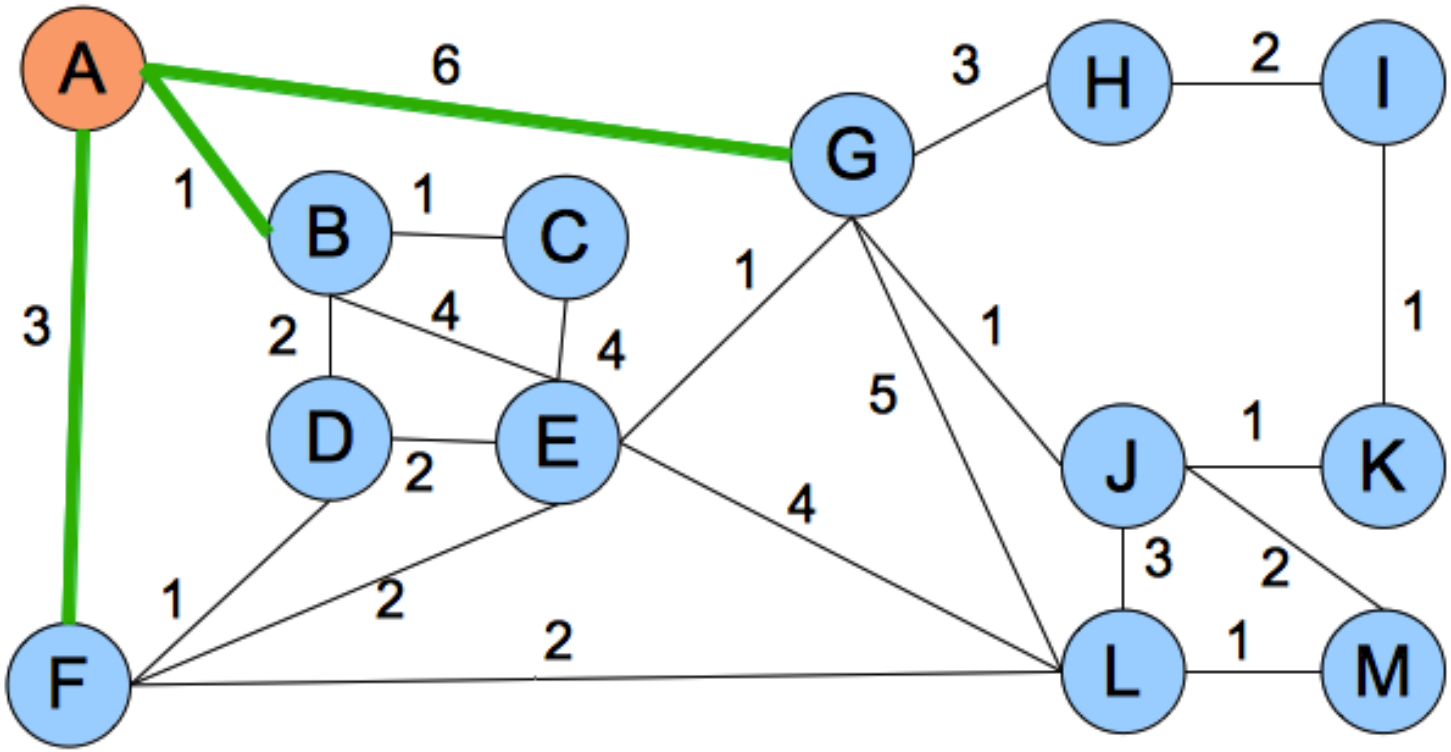

Figure: Spanning tree construction.

Prim-Jarnik: Minimal spanning tree

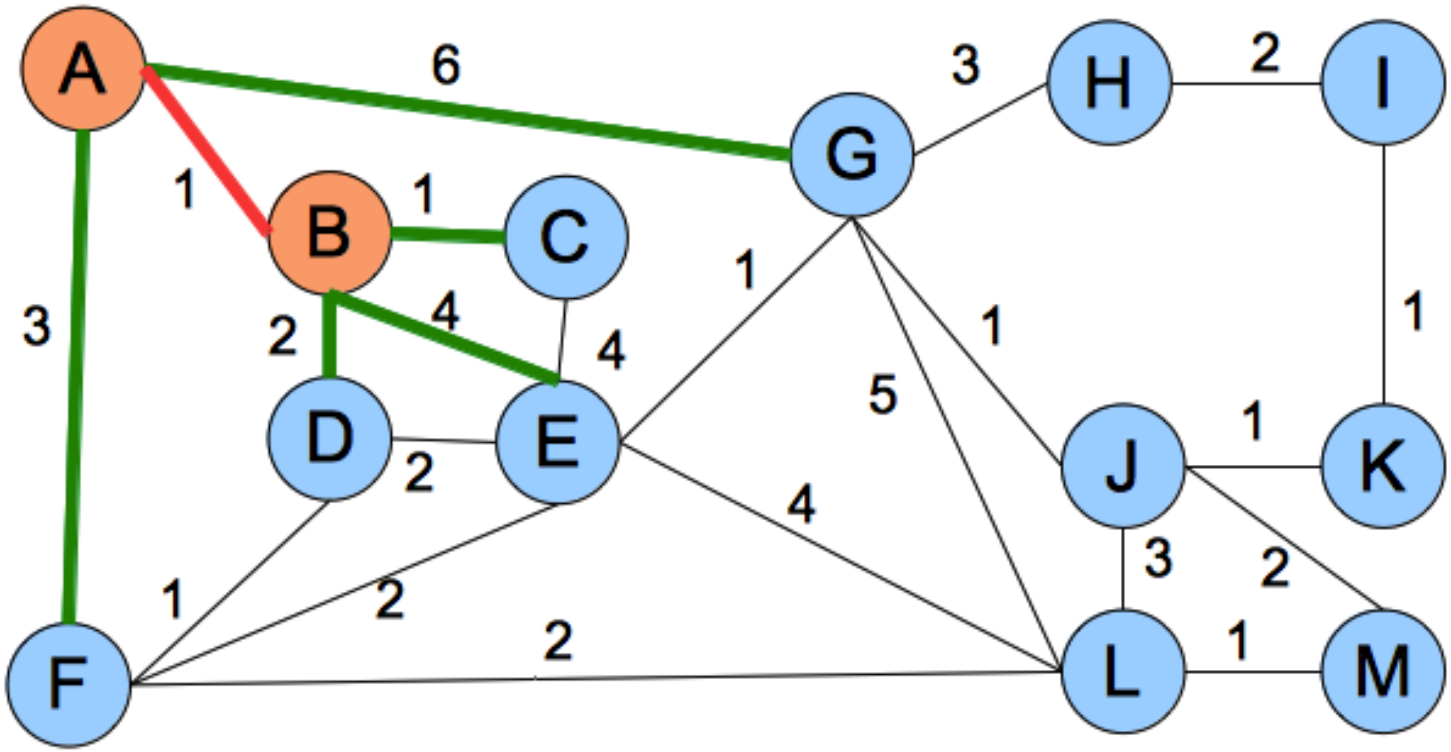

Figure: Spanning tree construction.

Prim-Jarnik: Minimal spanning tree

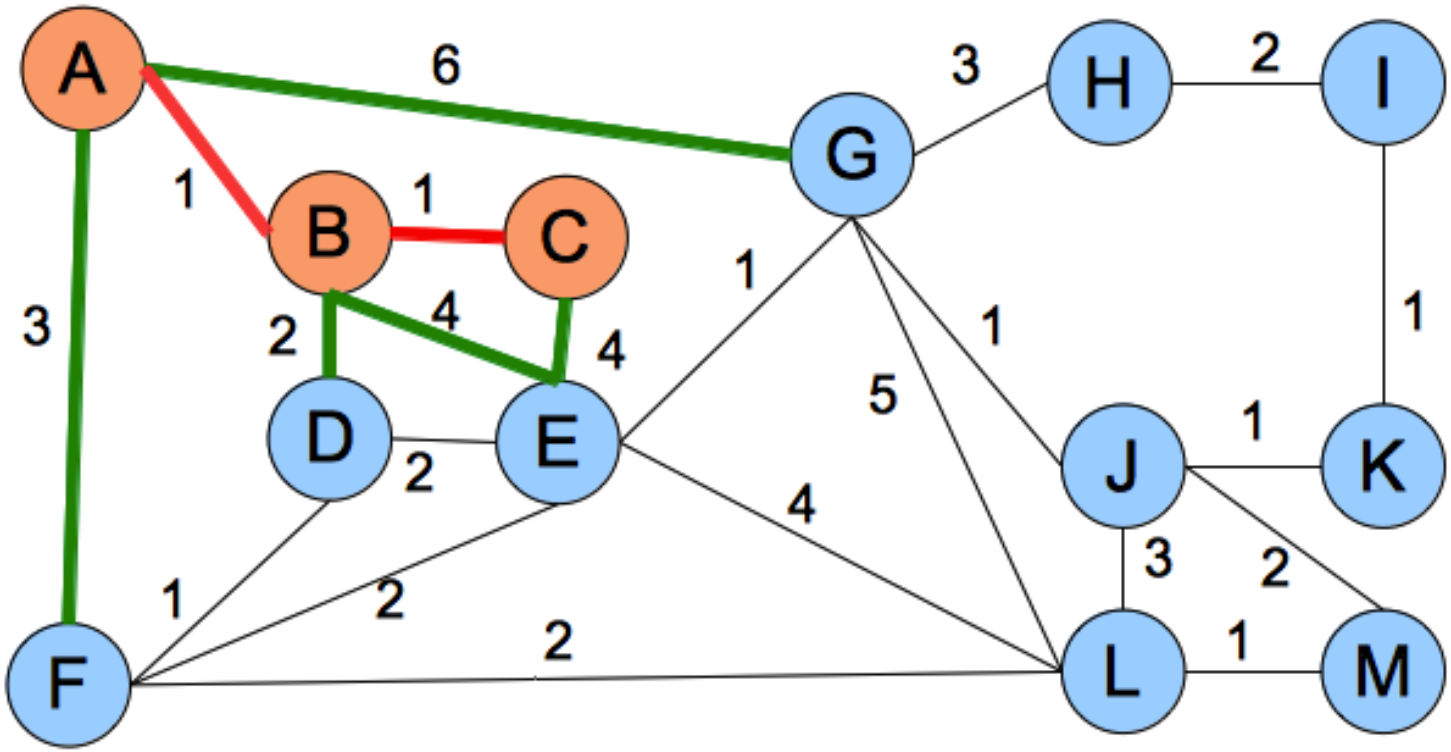

Figure: Spanning tree construction.

Prim-Jarnik: Minimal spanning tree

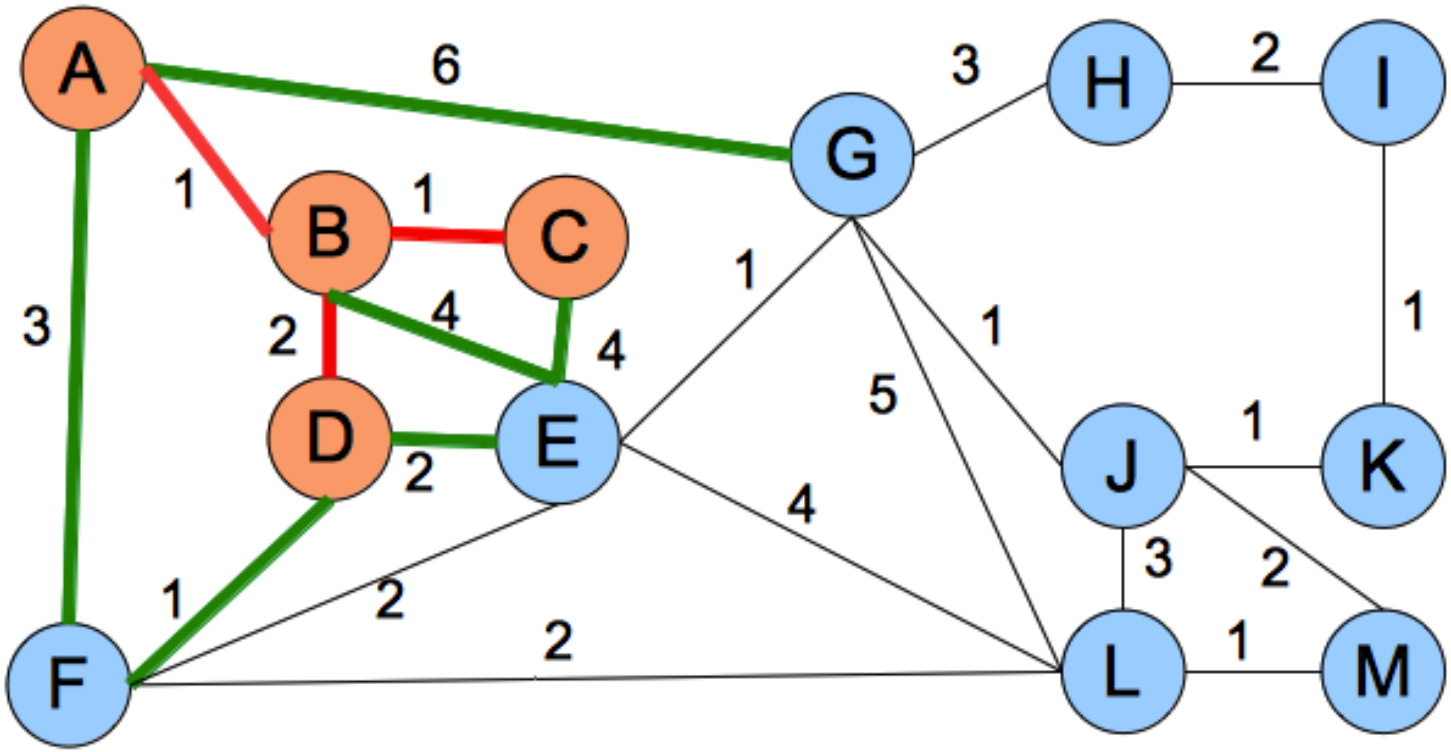

Figure: Spanning tree construction.

Prim-Jarnik: Minimal spanning tree

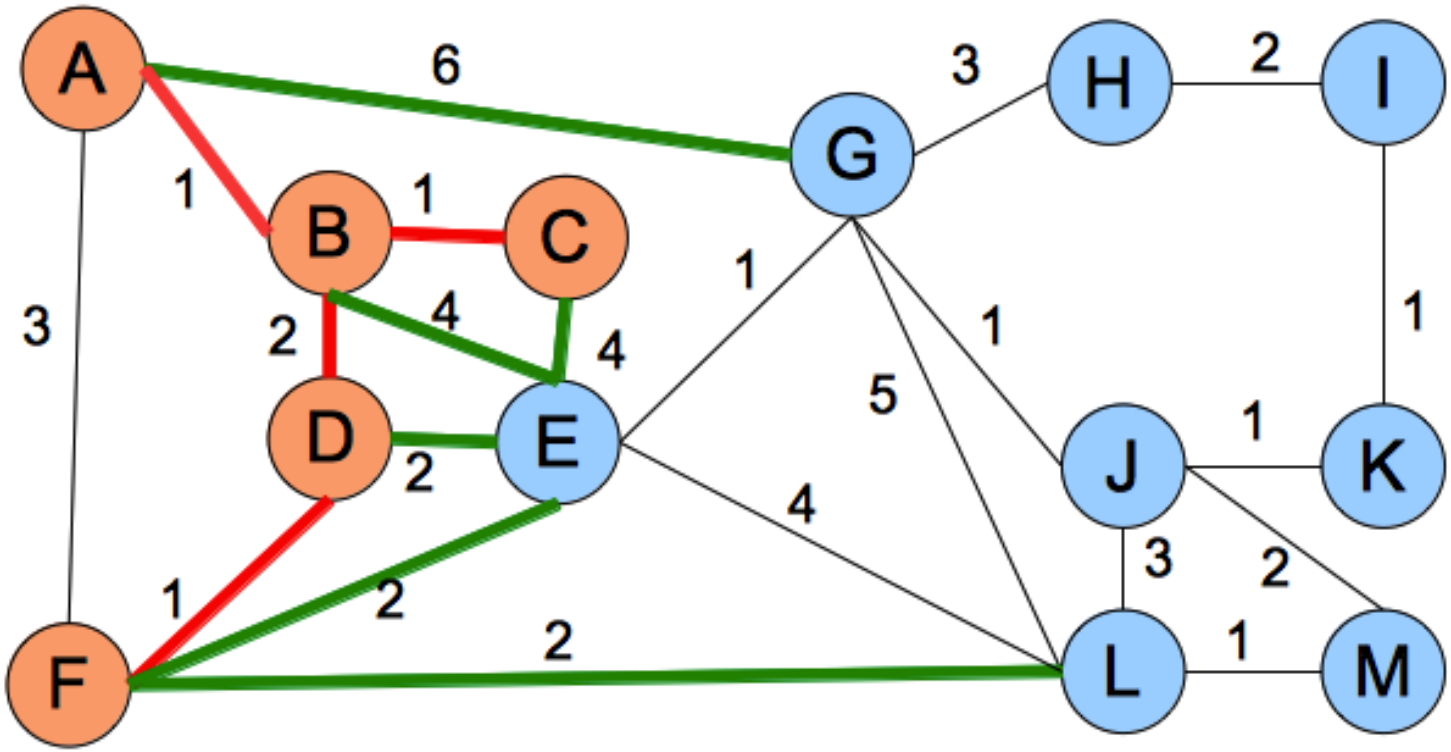

Figure: Spanning tree construction.

Prim-Jarnik: Minimal spanning tree

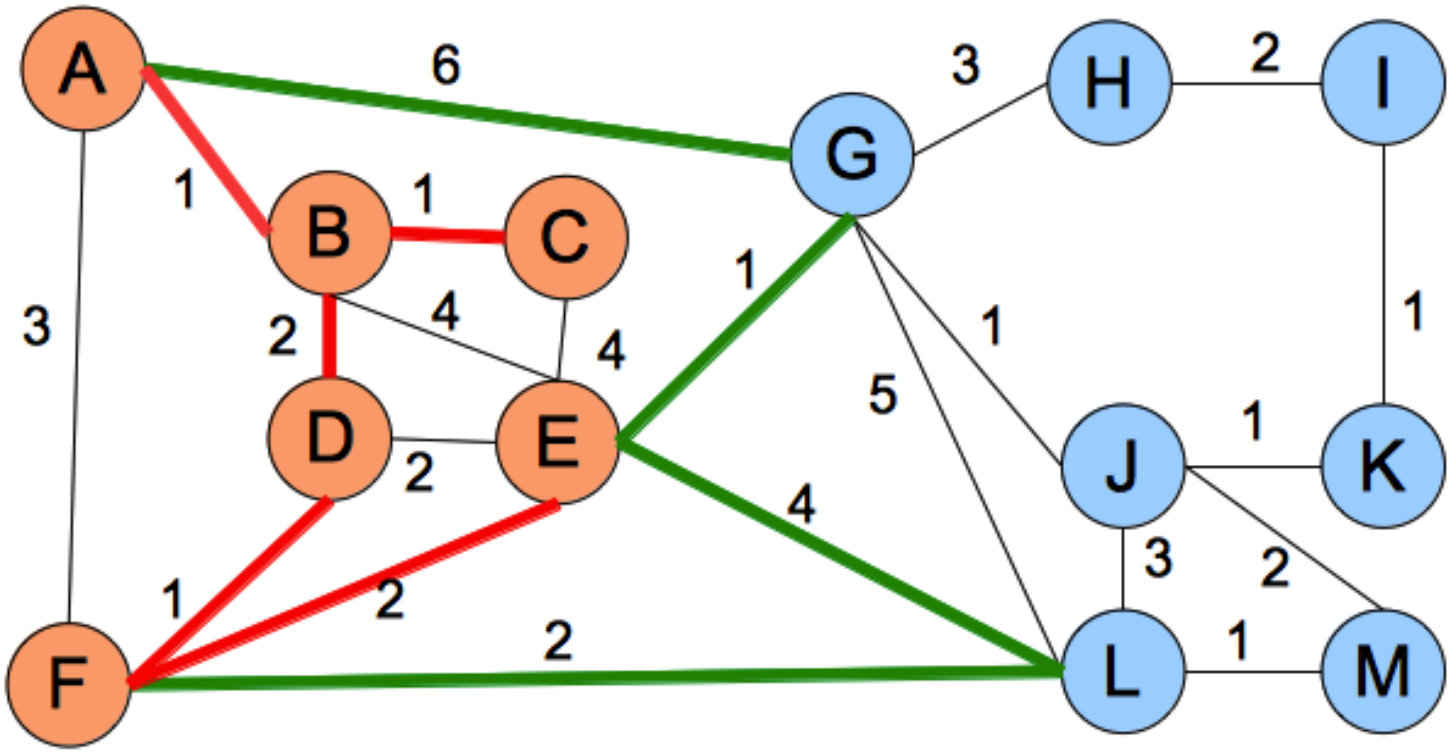

Figure: Spanning tree construction.

Prim-Jarnik: Minimal spanning tree

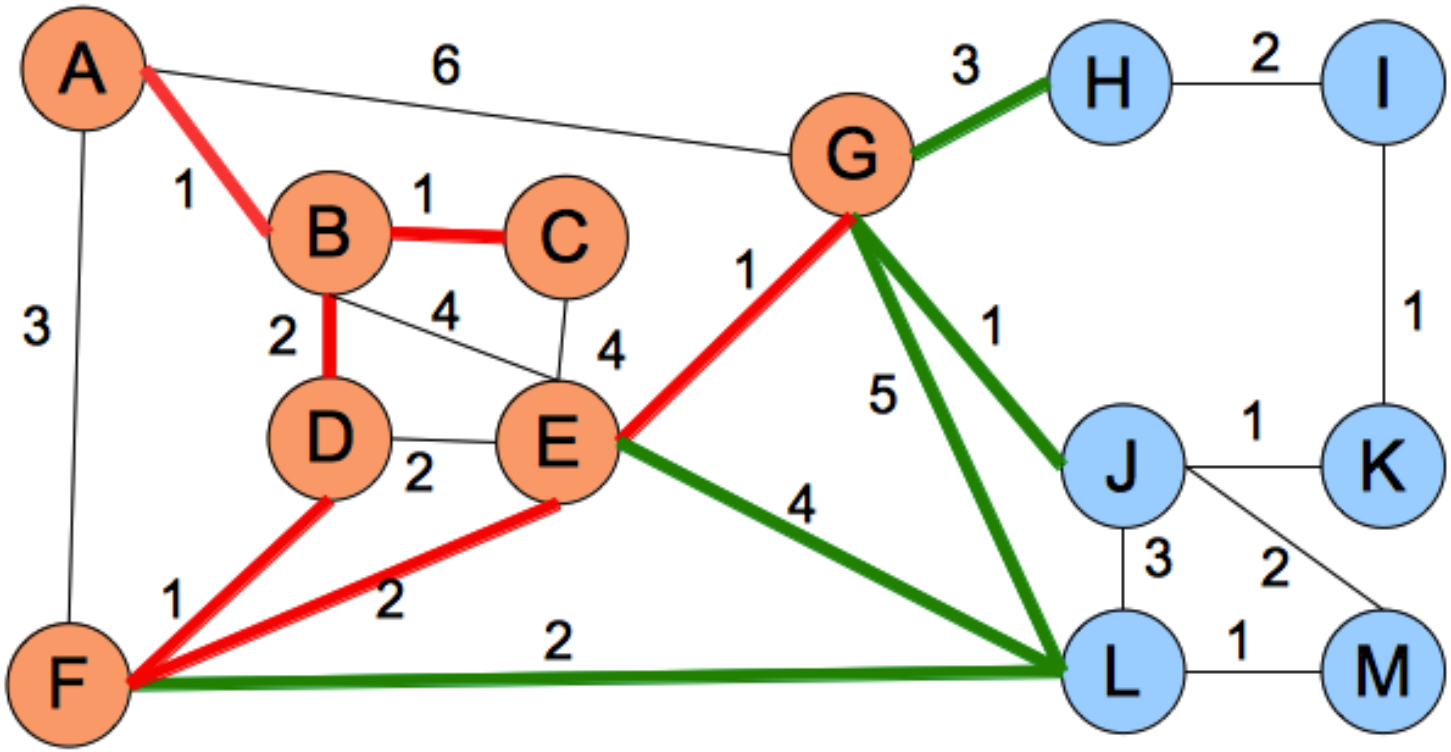

Figure: Spanning tree construction.

Prim-Jarnik: Minimal spanning tree

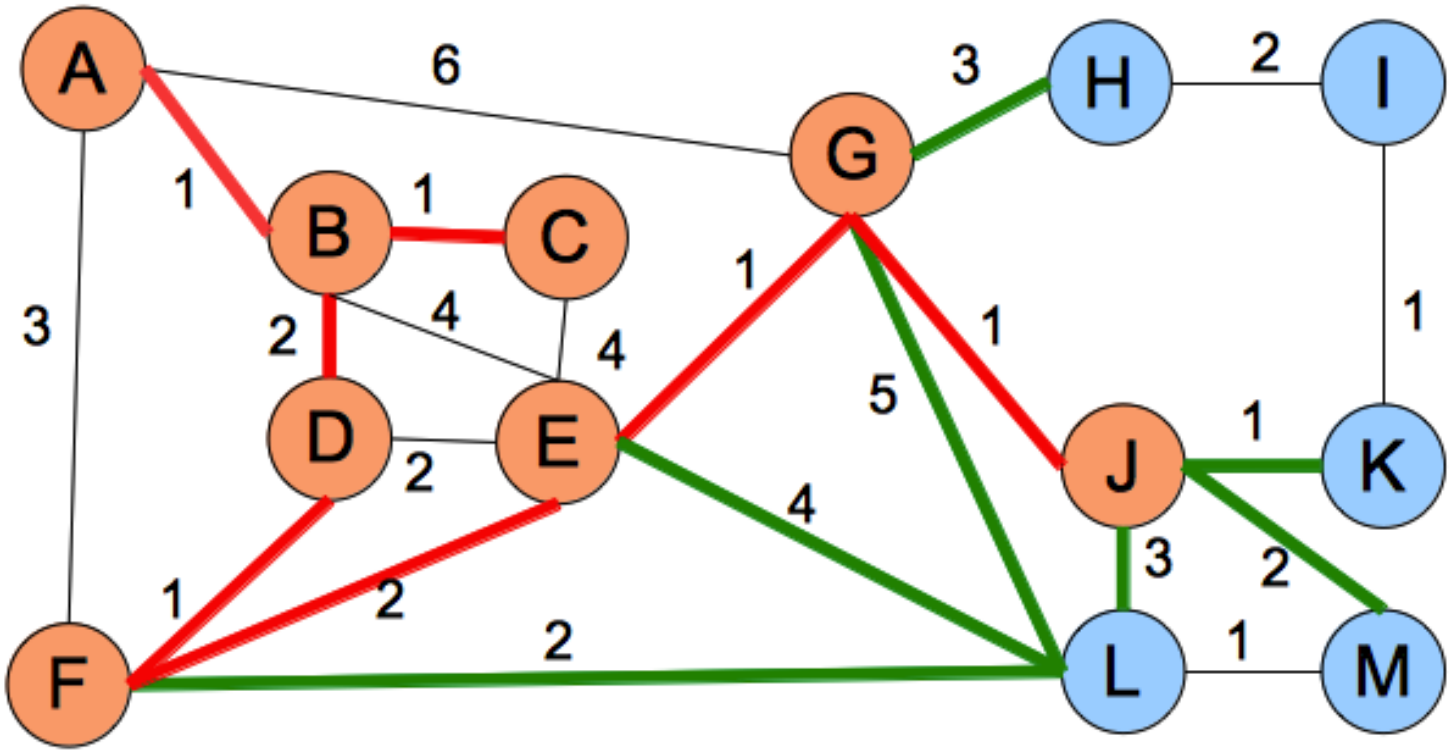

Figure: Spanning tree construction.

Prim-Jarnik: Minimal spanning tree

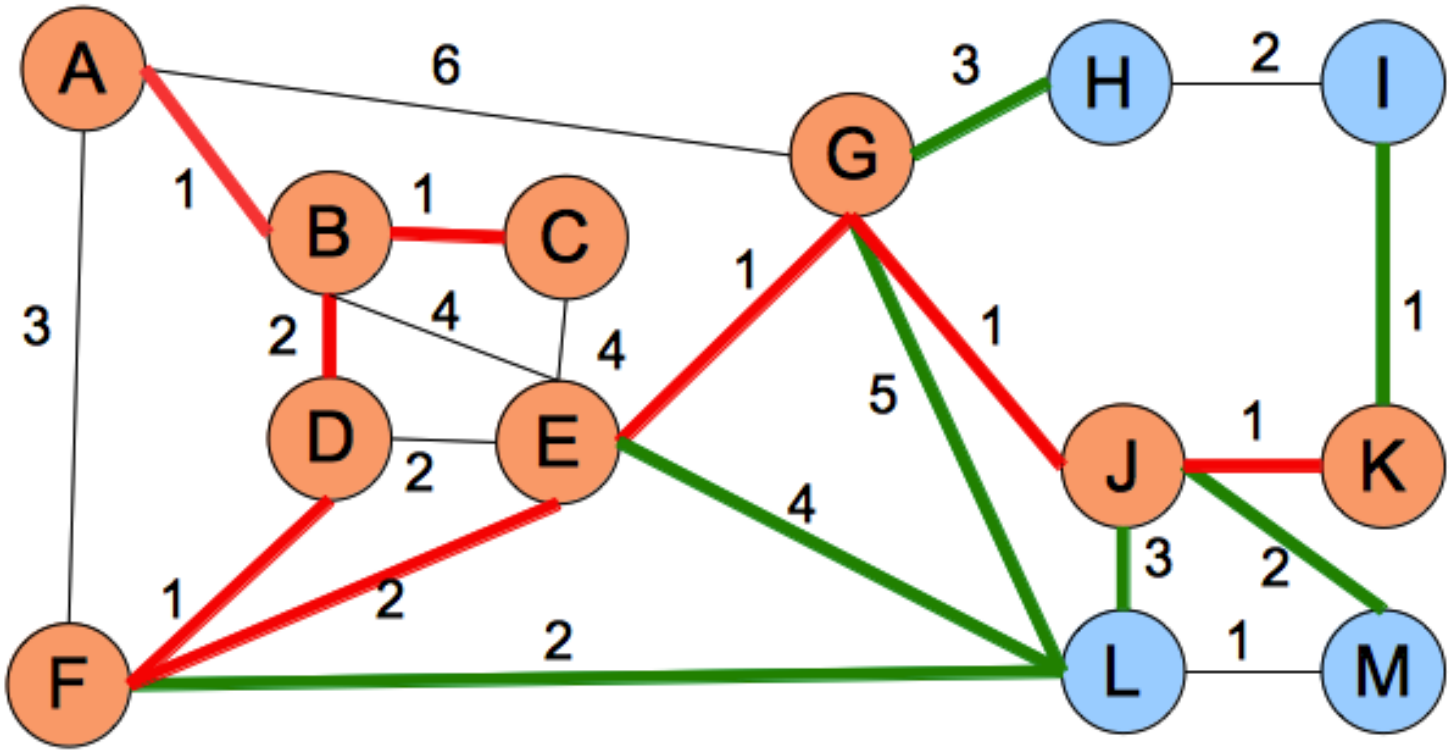

Figure: Spanning tree construction.

Prim-Jarnik: Minimal spanning tree

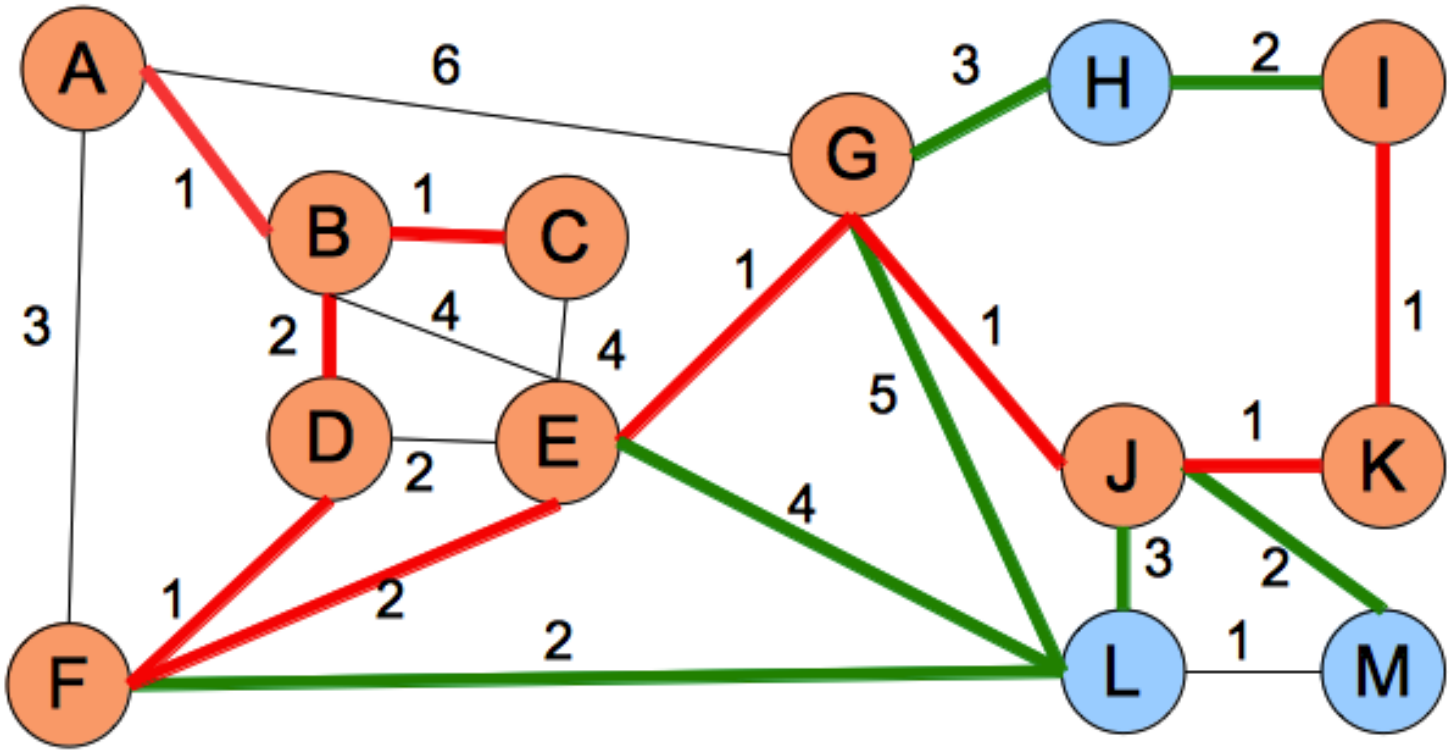

Figure: Spanning tree construction.

Prim-Jarnik: Minimal spanning tree

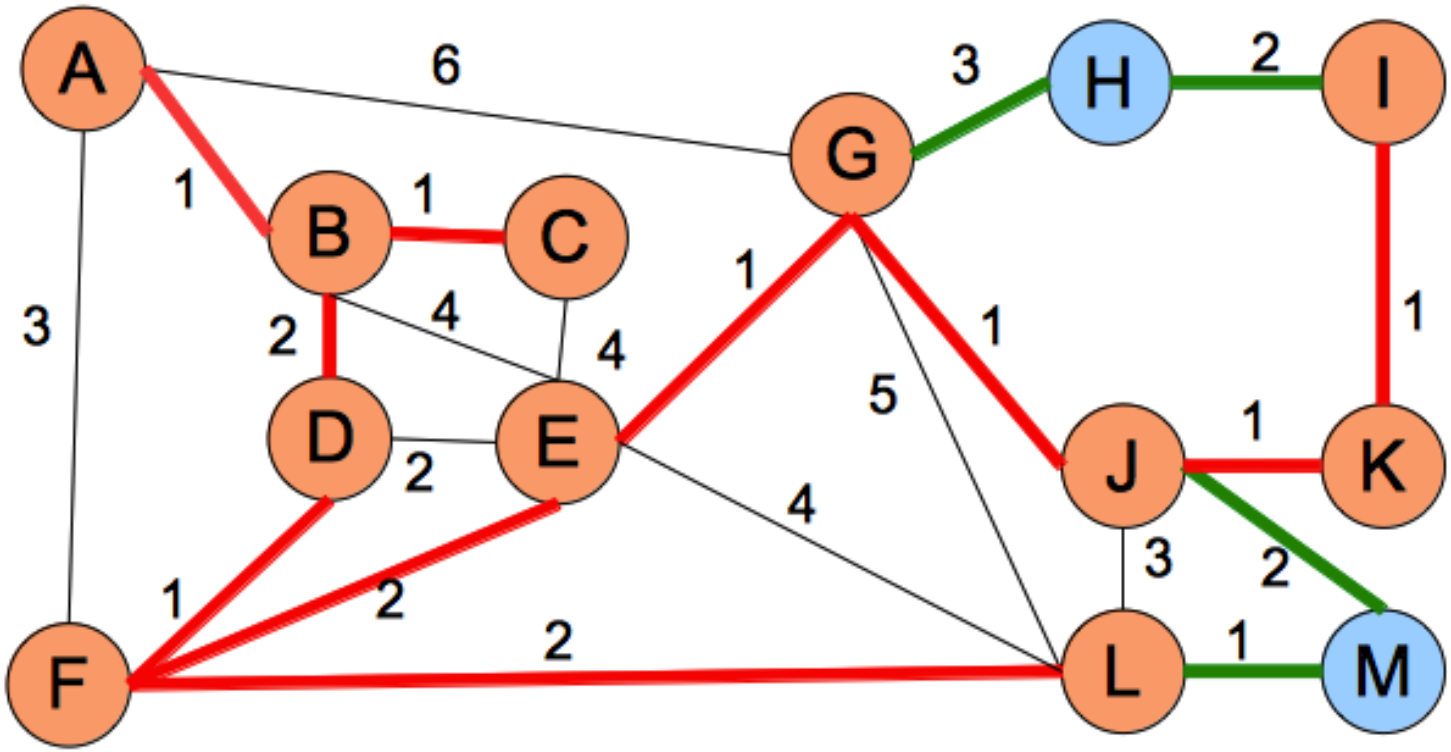

Figure: Spanning tree construction.

Prim-Jarnik: Minimal spanning tree

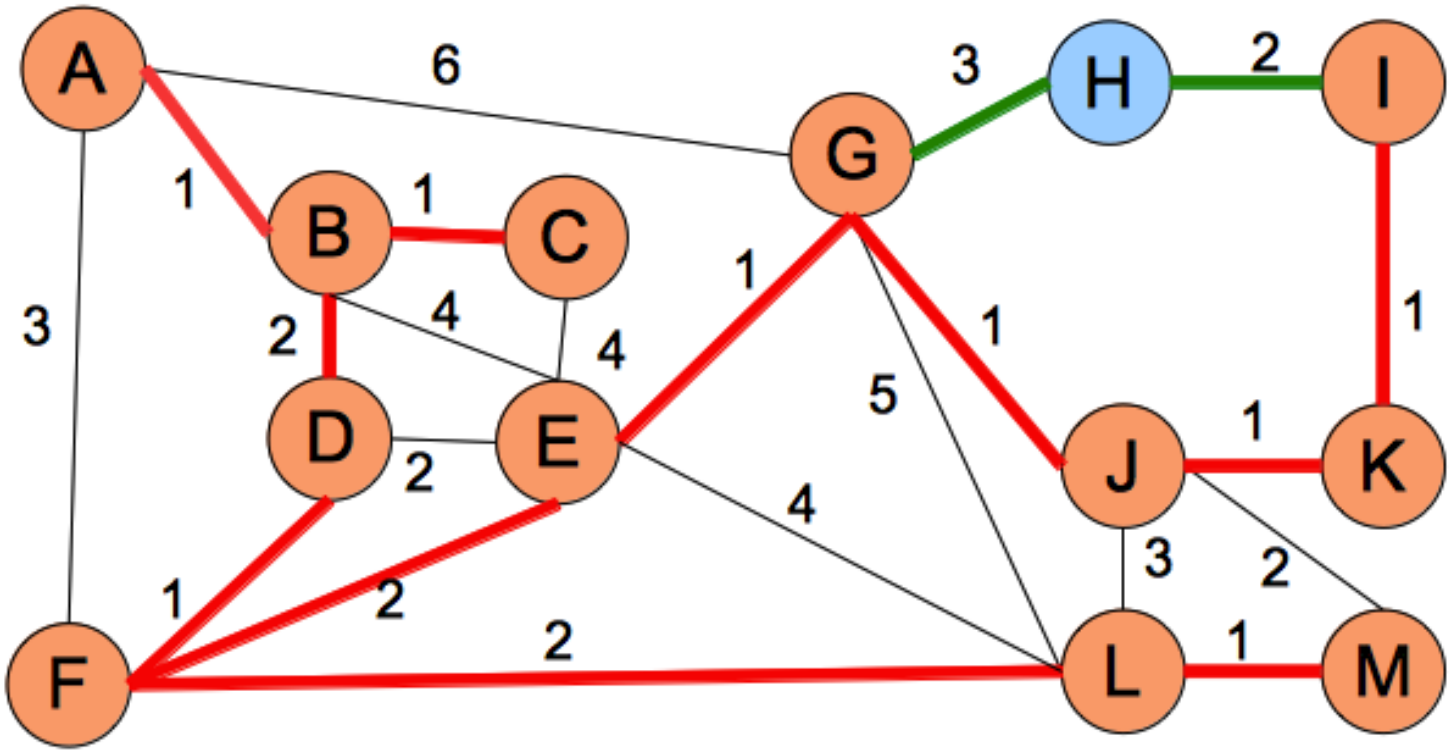

Figure: Spanning tree construction.

Prim-Jarnik: Minimal spanning tree

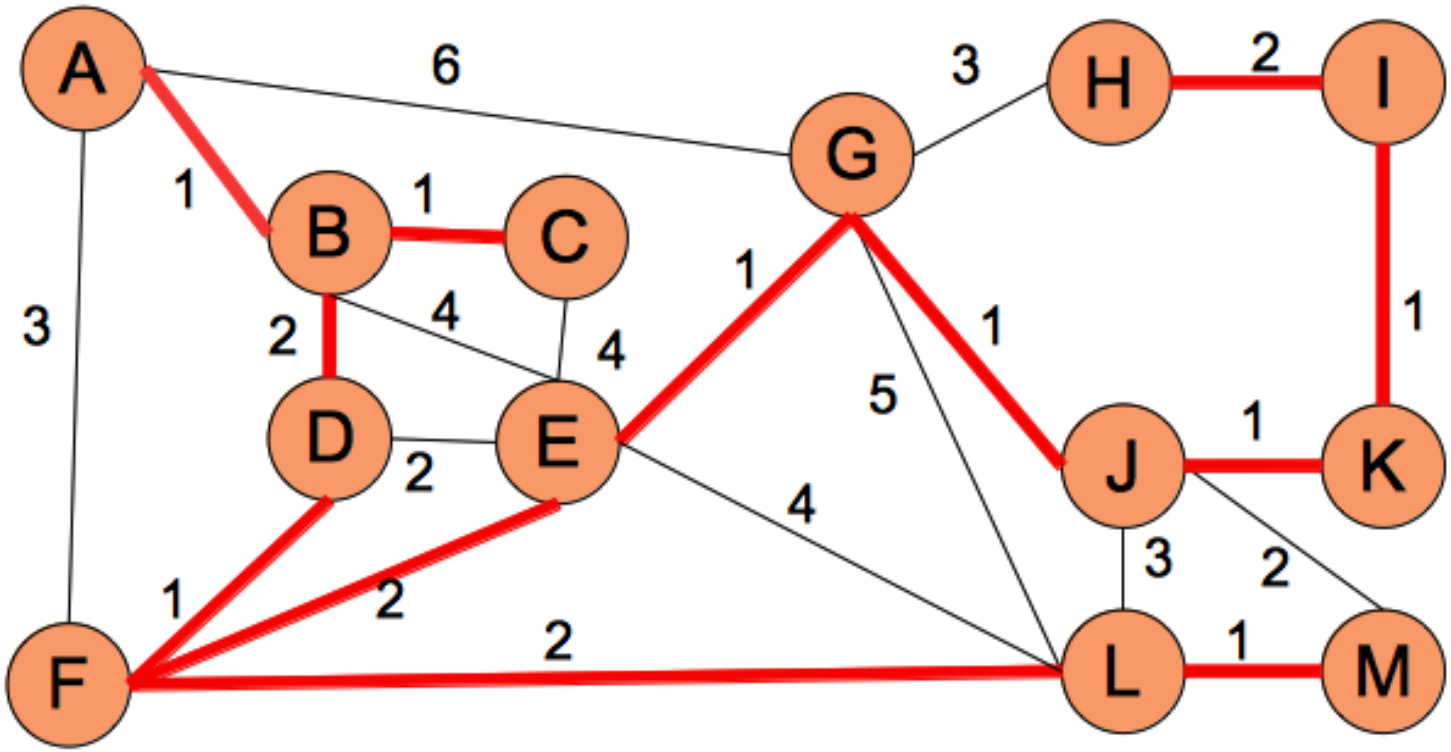

Figure: Spanning tree construction.

Implementation: Prim-Jarnik: Minimal spanning tree

- It is basically like a priority search
- Implementation uses a priority queue:
  - Nodes on edges with smallest weight are visited first
  - Cycles are checked and corresponding edges removed
  - Remove from queue edges becoming superfluous (joining spanning tree nodes)
- Add minimal edge not forming cycle to spanning tree
- Similar to OS time sharing queues: examine all nodes neighboring current tree
- As before 3 types of nodes: processed, in queue and untouched
- queue nodes are borders of up to date spanning tree
- Efficiency: O((E+V)log V)

Kruskal Method: Minimal spanning tree

- Different principle:
  - Add edges successively, and use shortest edge not building a cycle
  - This means building bottom up: join subtrees until a single tree is left
- Implementation based priority queue, and checking if a cycle is built
- Pretty efficient, works in O(E log E)

Kruskal Method: Minimal spanning tree

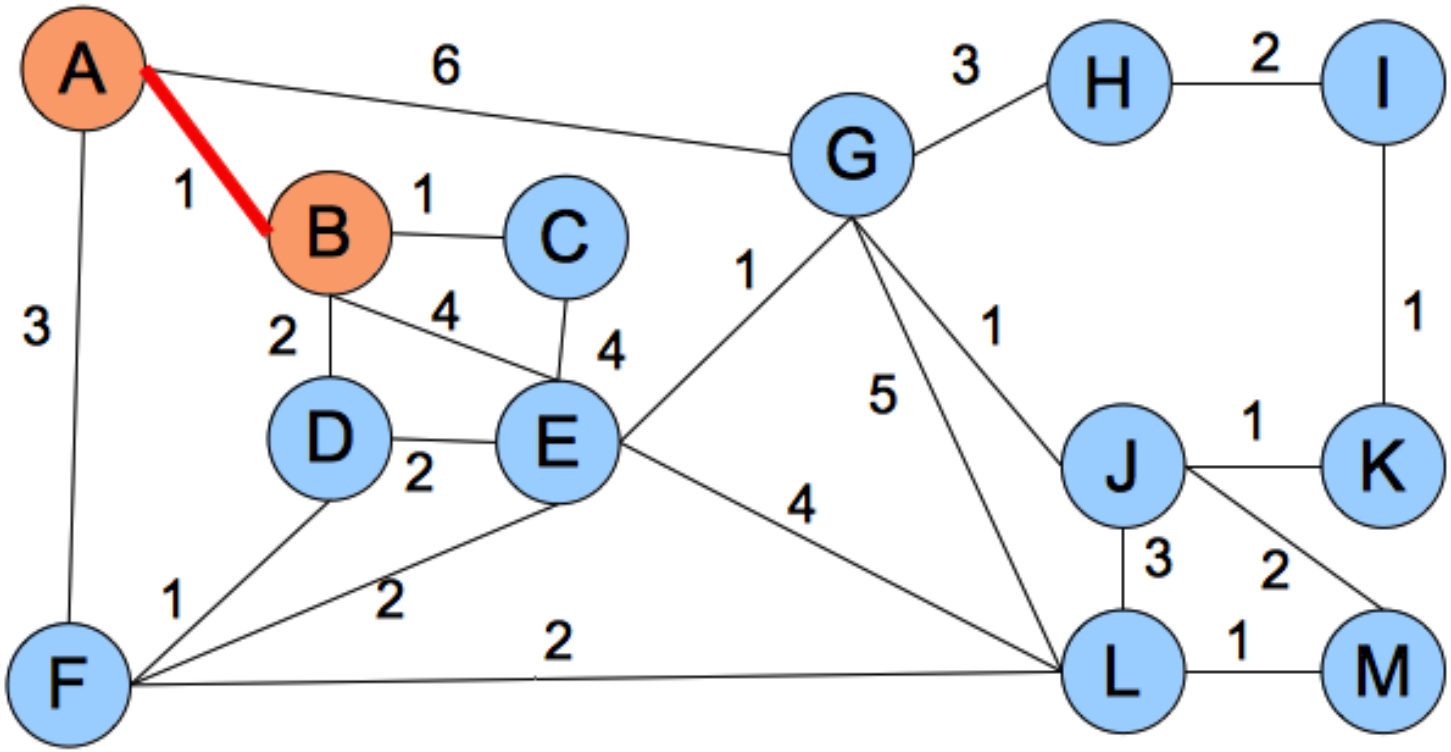

Figure: Spanning tree construction with the Kruskal method.

Kruskal Method: Minimal spanning tree

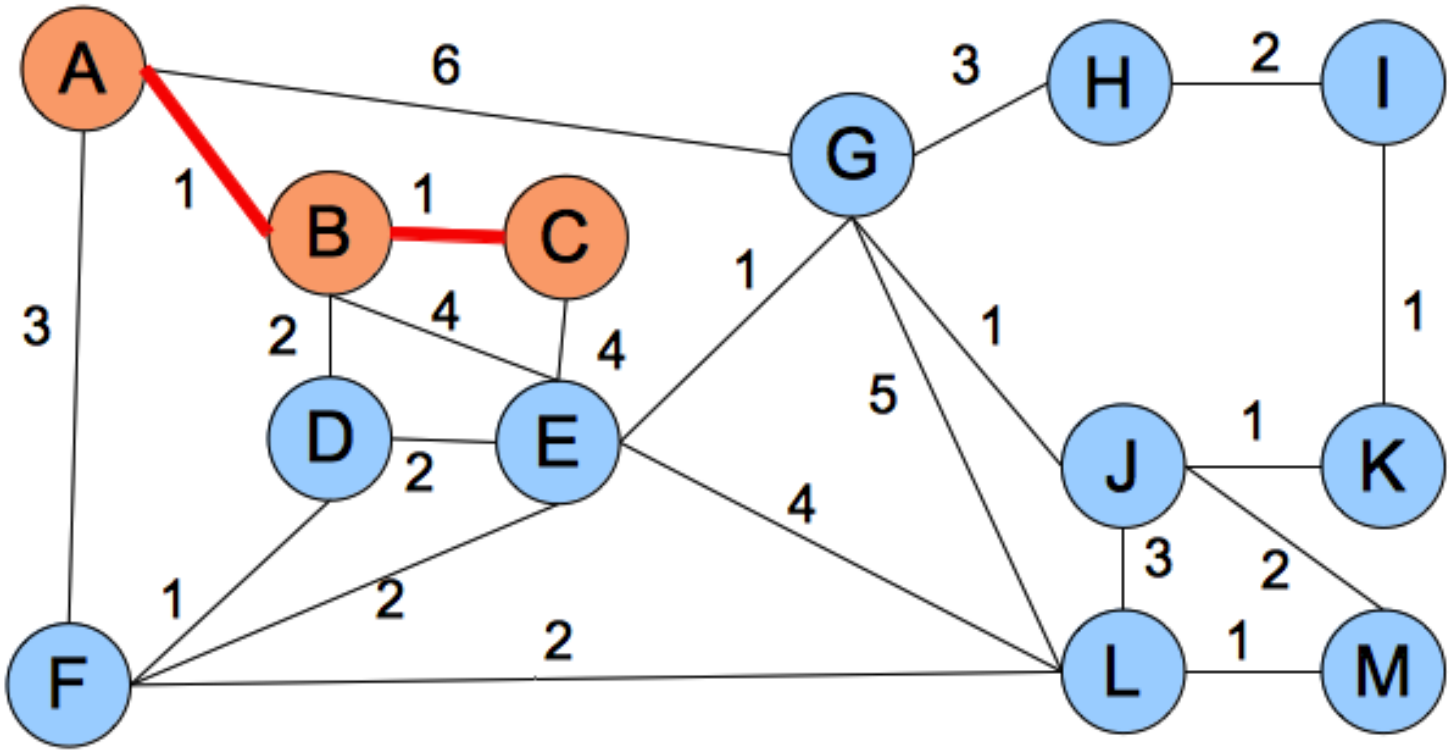

Figure: Spanning tree construction with the Kruskal method.

Kruskal Method: Minimal spanning tree

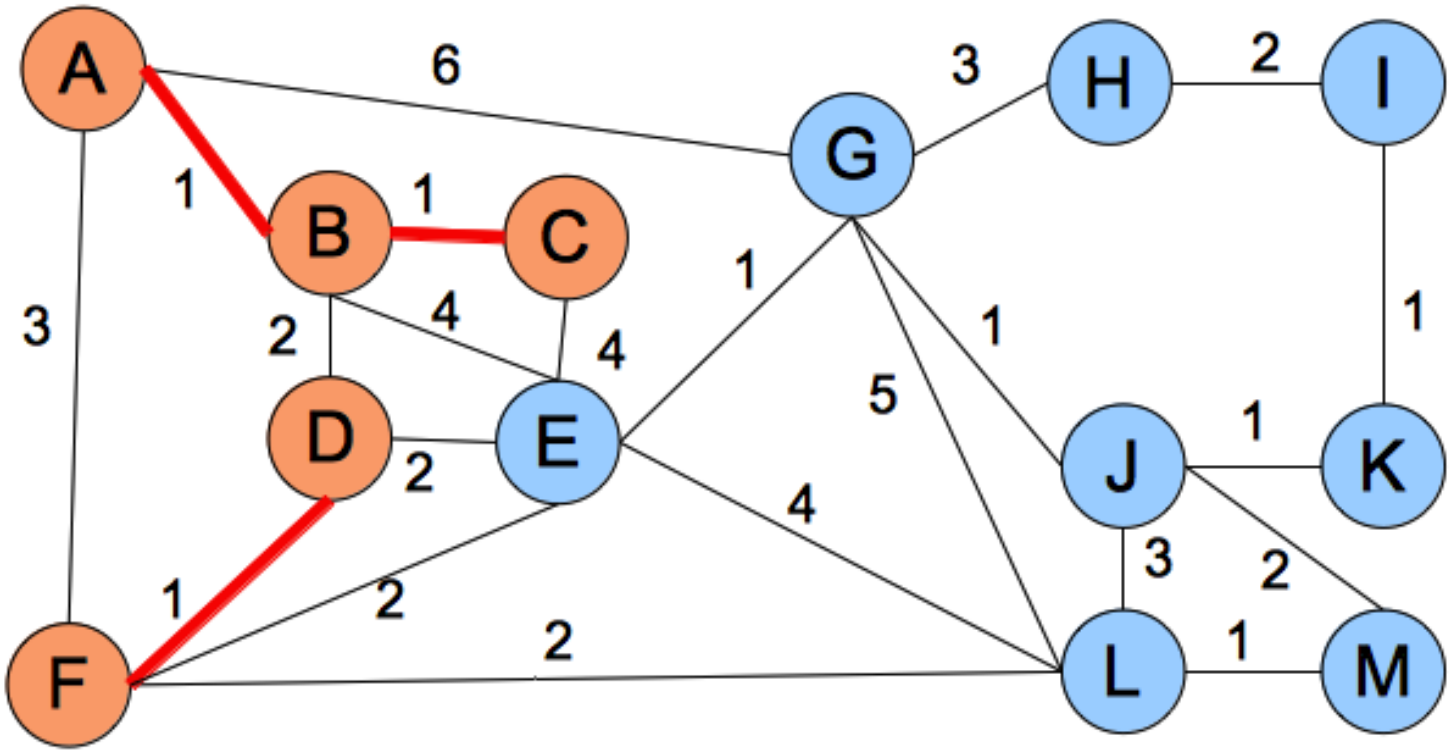

Figure: Spanning tree construction with the Kruskal method.

Kruskal Method: Minimal spanning tree

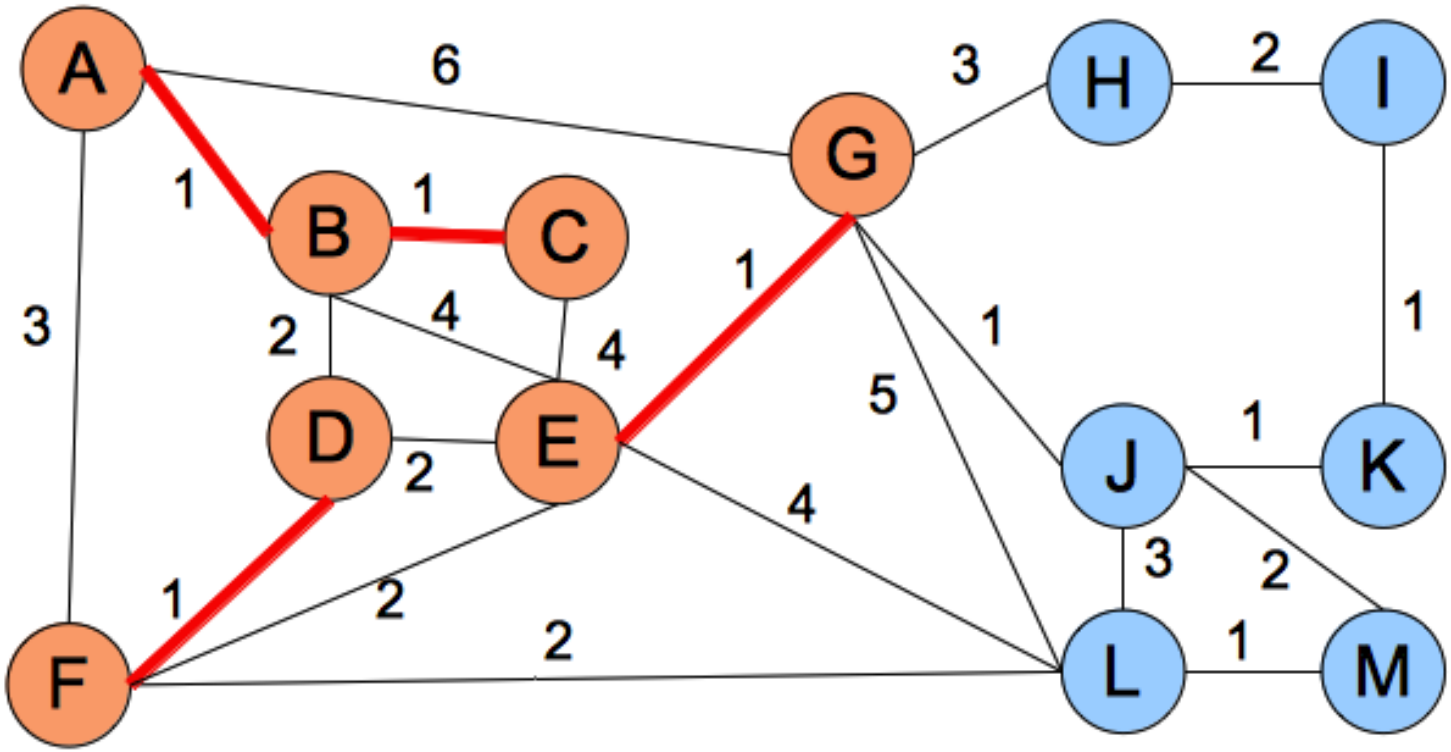

Figure: Spanning tree construction with the Kruskal method.

Kruskal Method: Minimal spanning tree

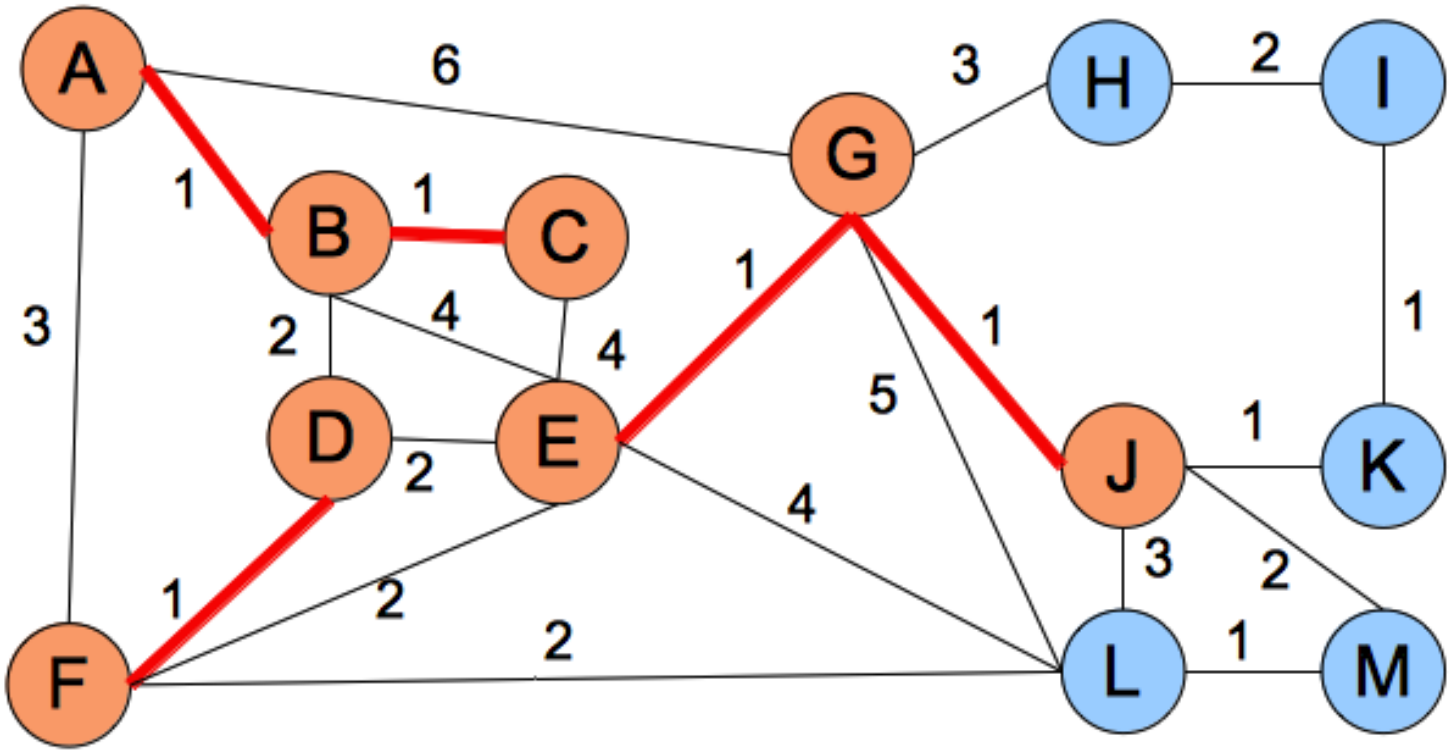

Figure: Spanning tree construction with the Kruskal method.

Kruskal Method: Minimal spanning tree

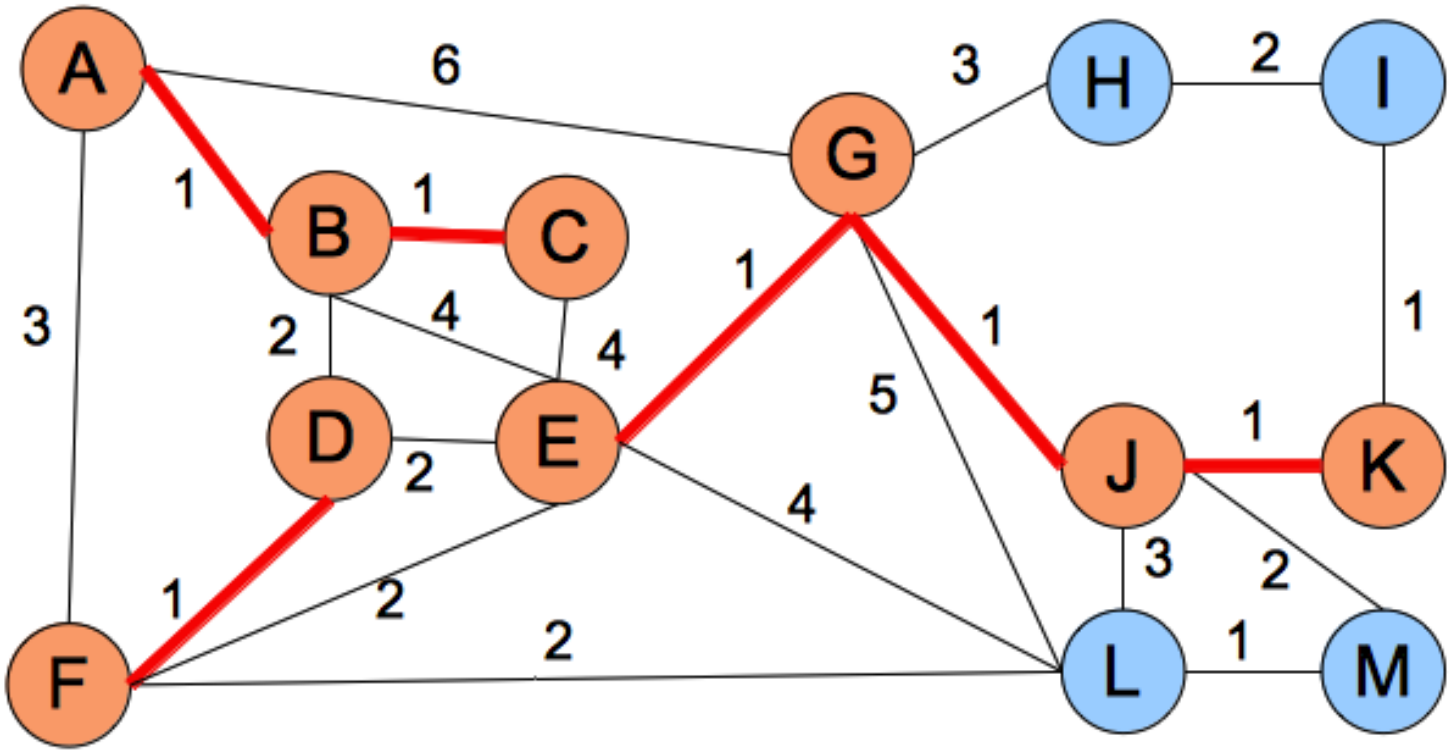

Figure: Spanning tree construction with the Kruskal method.

Kruskal Method: Minimal spanning tree

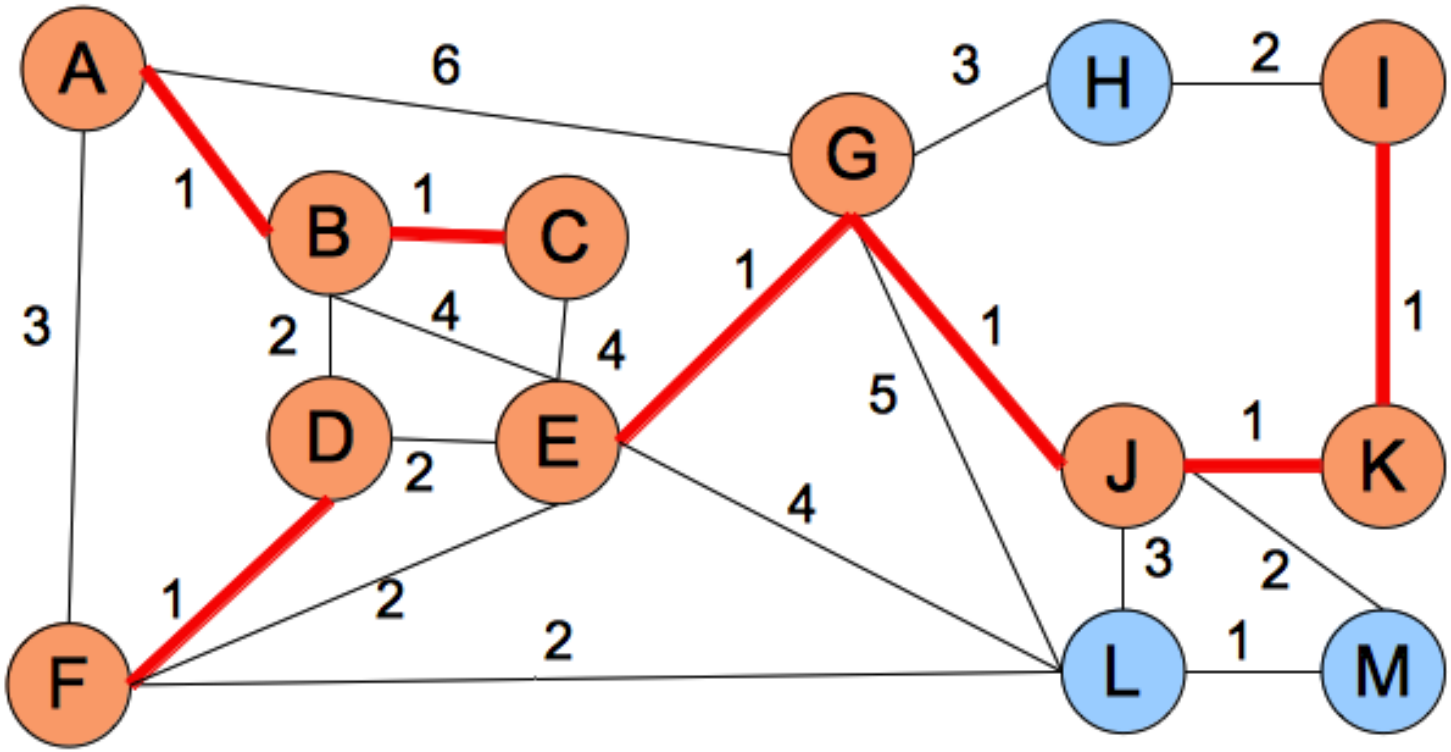

Figure: Spanning tree construction with the Kruskal method.

Kruskal Method: Minimal spanning tree

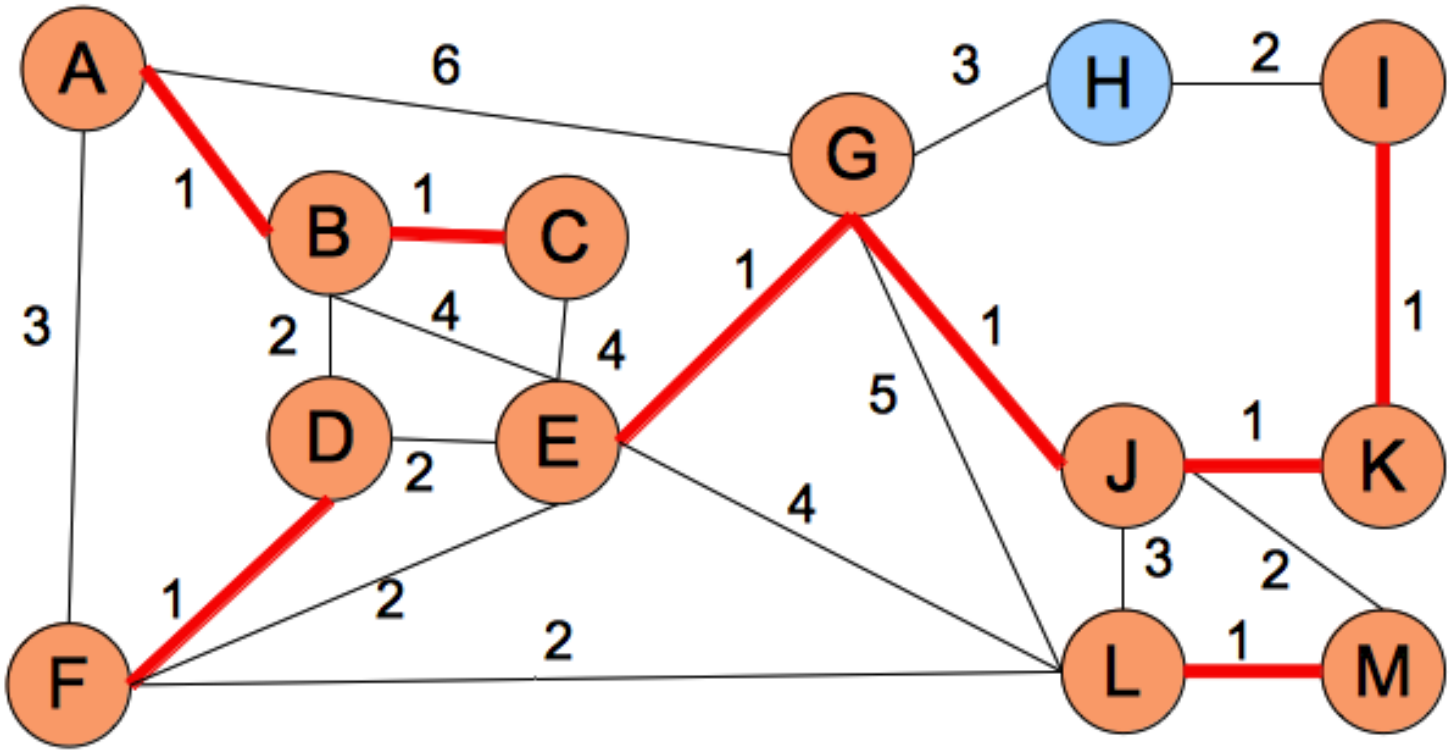

Figure: Spanning tree construction with the Kruskal method.

Kruskal Method: Minimal spanning tree

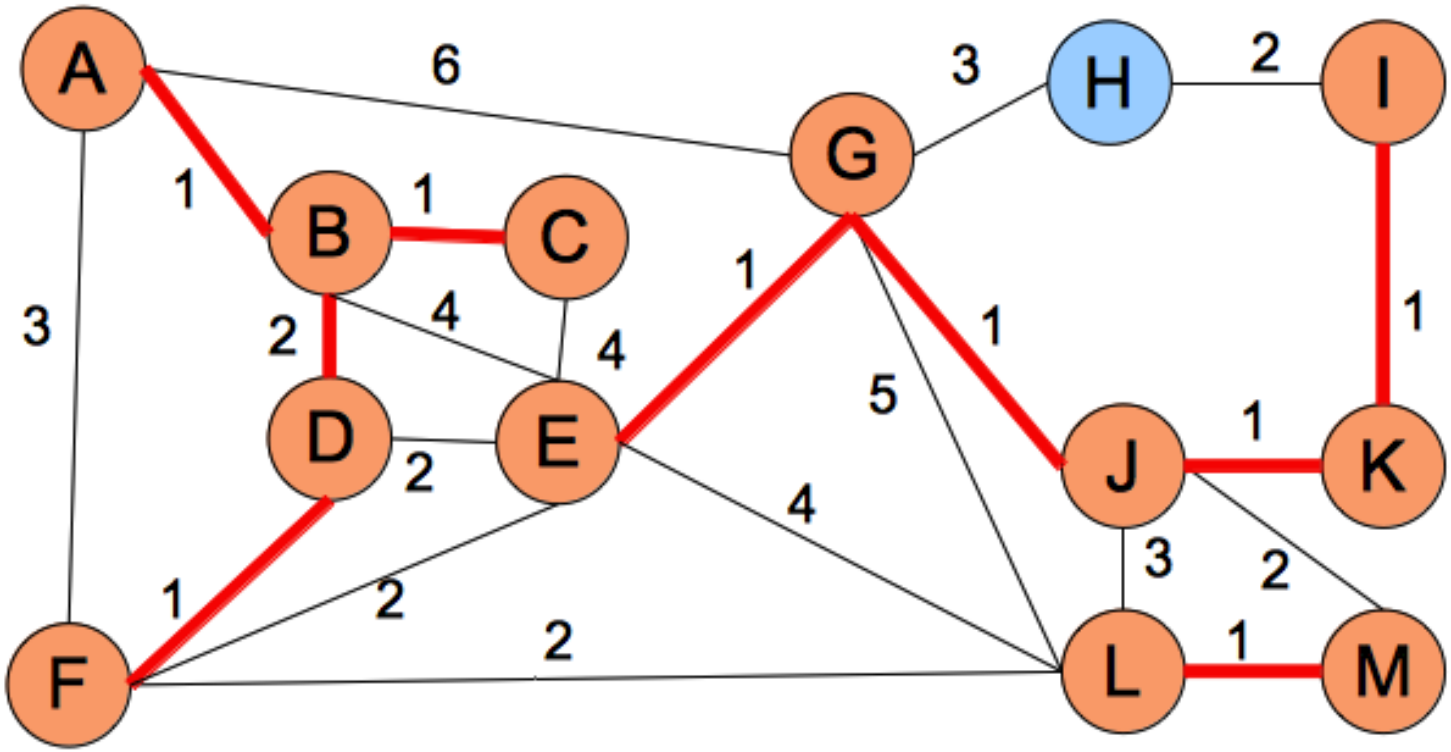

Figure: Spanning tree construction with the Kruskal method.

Kruskal Method: Minimal spanning tree

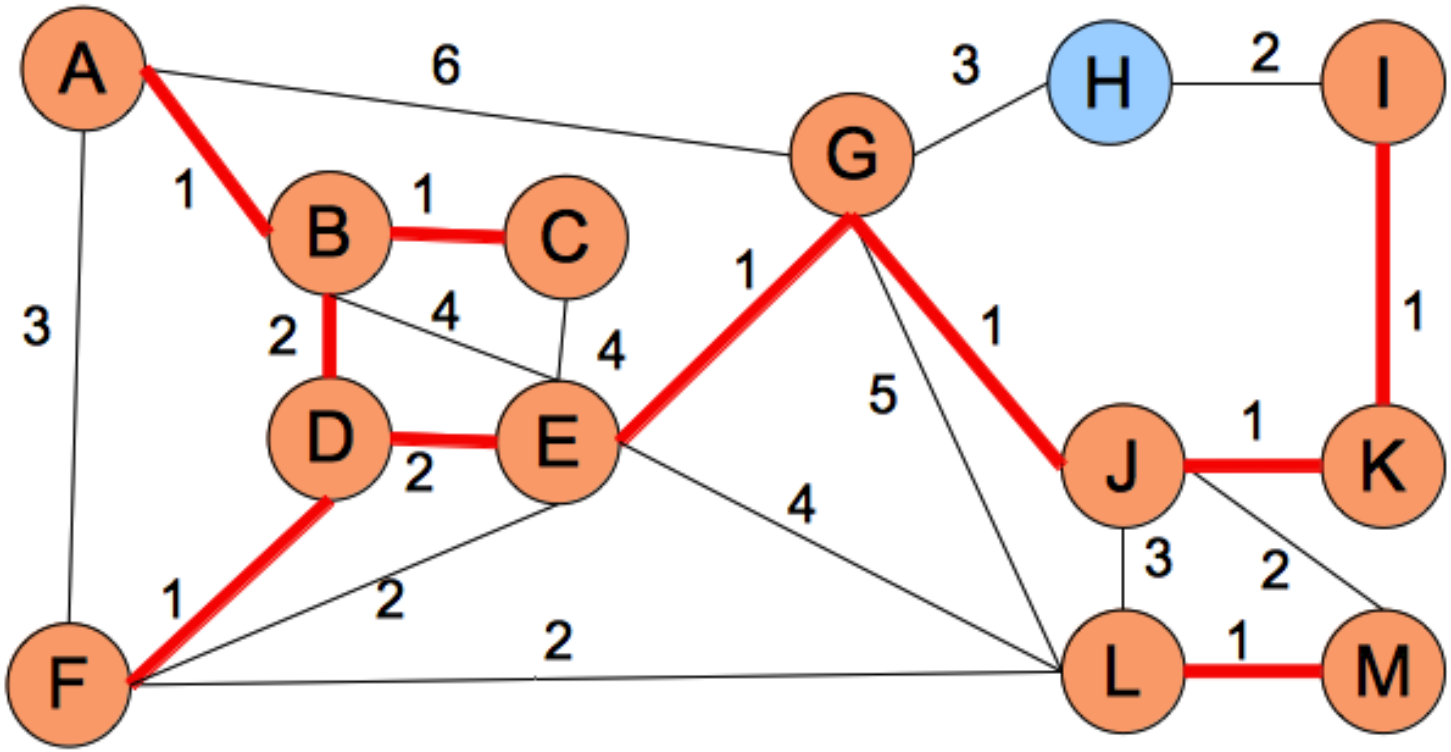

Figure: Spanning tree construction with the Kruskal method.

Kruskal Method: Minimal spanning tree

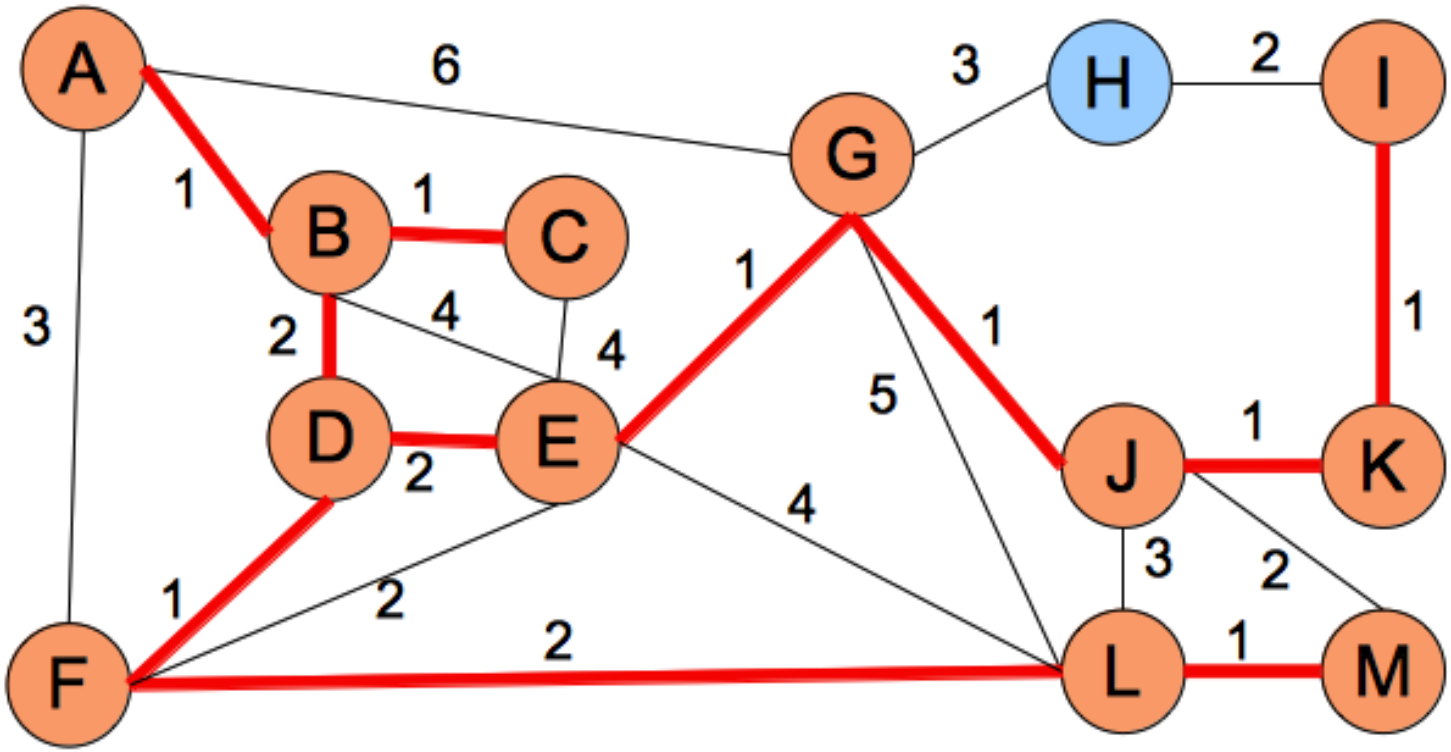

Figure: Spanning tree construction with the Kruskal method.

Kruskal Method: Minimal spanning tree

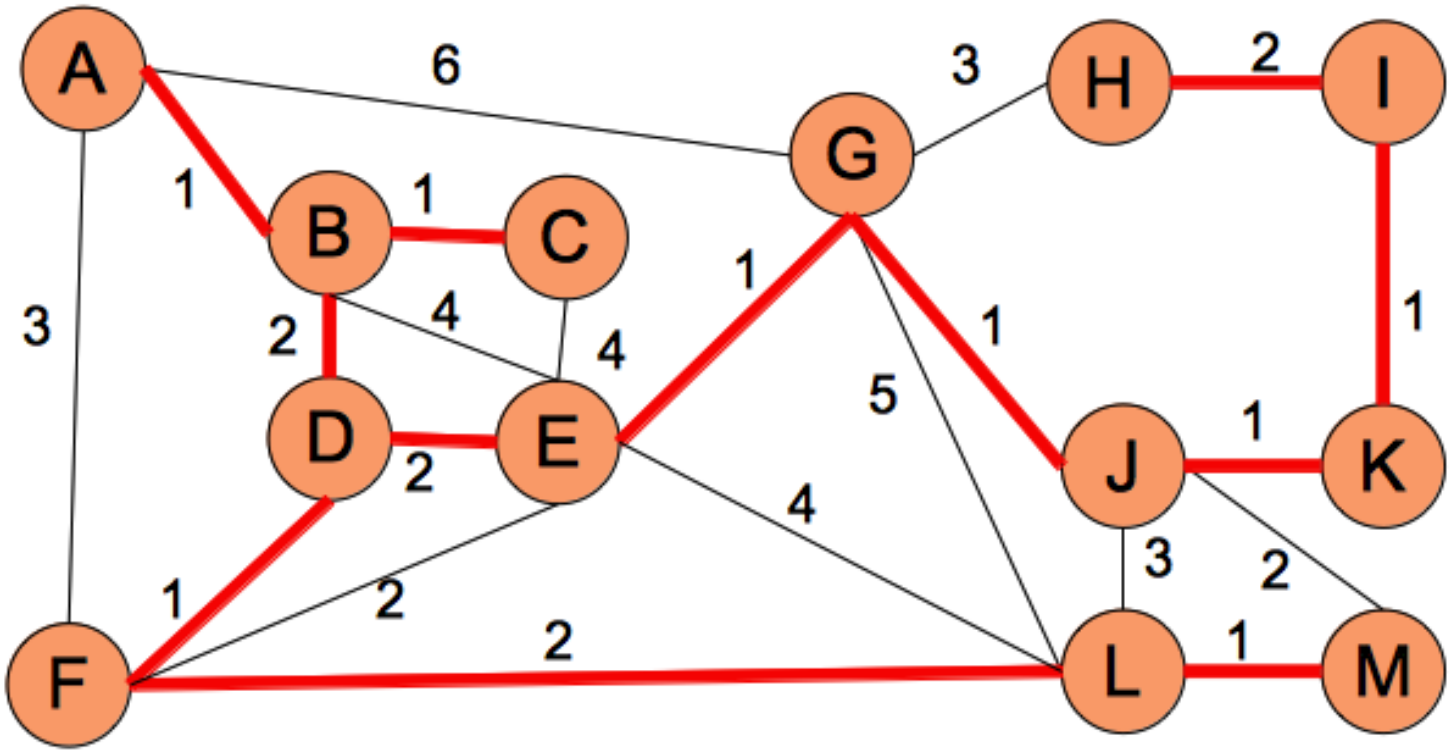

Figure: Spanning tree construction with the Kruskal method.

Prims algorithm: Minimal spanning tree

- Here, matrix representation convenient
- Searching through neighbors slightly more complicated (remember smallest)
- Use of a priority queue storing corresp. nodes and their values (external to matrix) and implemented in a vector
- Every time a node is removed from border, priority -> list updated and smallest registered
- This allows combining 2 ops. in 1 step , therefore the algo. works in linear time

Prims algorithm: Minimal spanning tree

- Worst case efficiency: (V^2)
- By dense graphs: O(V)
- Note that this algorithm basically equals:
  - For Matrix representation the Prim Algorithm for the minimal spanning tree
  - The Dijkstra algorithm for shortest path

Shortest path

- Shortest path
- In a non weighted graph, this is the shortest path (with less edges) (weights=1)
- Solving algorithm for unweighted graphs: breadth search starting from start node!!
  - Since this is like level-order traversal, first hit is the shortest path
- General case: priority search does the job (hidden problem: minimal spanning tree)

Flux in a network

- Problem:
  - Given a net (of pipelines), find switch positions so that maximal flux can be pumped
- Def: A network is a directed graph in which there are 2 special nodes:
  - One node that has no incoming arcs (source)
  - One node that has no outgoing arcs (drain)
- The weights of the edges are called capacities of the edges
- A flux is a new distribution of the weights, such that each new weight is $<=$ capacity
- The value of a flux is the flux into (or out of) the network
- The flux problem in a network is to find out a flux with maximum value

Implementation: Flux in a network

- One can use adjacency Matrix, whereby each position has one value for the capacity and one for the flux
- Note that pipes can be filled in 2 directions:
  - whenever an edge x -> y exists with flux f and capacity s,
  - at the edge y -> x flux -f and capacity -s added
- All updates must take this into account

Ford Fulkerson: Flux in a network

- Take a path from source to drain
- Take minimal capacity on path: pipelines transport at least this
- Do same for all independent paths: not all edge capacities will be used to their max
- You can increase flux always IF not all paths from source to drain have all node capacities full, else you cannot increase cap. (maxflow/mincut theorem)

Pairing

- Bipartite graph: a graph so that the nodes can be partitioned in two parts in such a way that each edge links one subset with the other one
- For convenience, vertices in one partition are labeled with letters, and the others with numbers

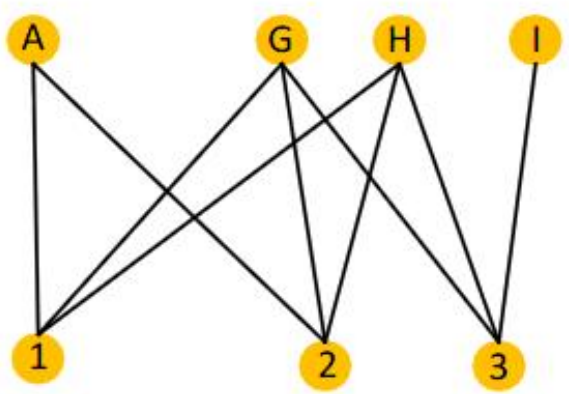

Figure: Bipartite graph.

Pairing

- Find biggest subset of pairs of numbers and letters such that no pairs contain the same letters or numbers
- Solution is done through flux in network, by adding source and drain at each edge and maximizing flux
- Edge weights are binary values

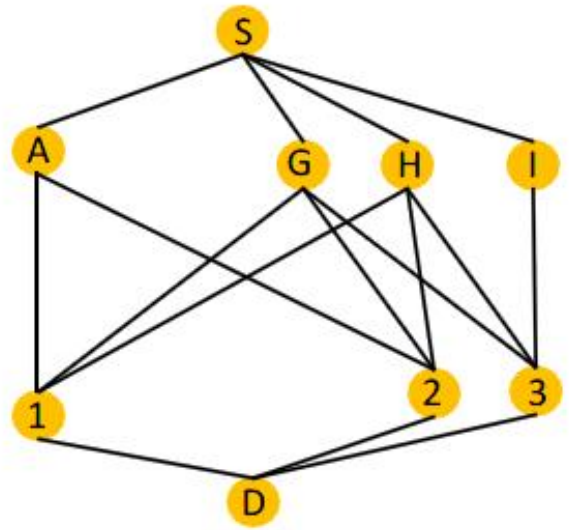

Figure: Bipartite graph with source and drain.

Stable marriage

- Solution is pretty simple
  - Let first man choose first woman according to his preference
  - Take 2nd man: if he prefers free woman, make contract
  - else use second in his preference List, and so on
  - IF woman already given, she chooses the one of the two men she prefers, and the abandoned candidate has to become again chooser, starting from where he was in his prefs list
- Runs in linear time