Chapter 8

Basic Graph Algorithms

8.1 Introduction

Graph theory has many practical applications in a variety of fields, including computer science, operations research, social sciences, and biology. Graphs come in numerous varieties. They may be *directed* or *undirected.* They may be *simple* or with loops and parallel edges. They may be *weighted* or *unweighted.* They may be *acyclic* or not. For example, when modeling a transportation networks, the nodes may represent important milestones in a city. A directed edge may represent an one-way street and an undirected edge may represent a street that can be traversed in either direction. The weights may represent the distance between the nodes.

In this chapter we cover some basic traversal algorithms in a simple directed graph. Section 8.2 gives a parallel LLP algorithm to traverse a graph. Section 8.3 gives a parallel LLP algorithm to construct a breadth-first-search (BFS) tree from a graph given a source vertex. Section 8.5 gives a parallel LLP algorithm to topologically sort a directed acyclic graph. Section 8.6 gives an LLP algorithm to find connected components in an undirected graph. Section 8.7 gives a faster LLP algorithm based on pointer jumping.

8.2 General Traversal in a Graph

Given a graph (V, E) and a source vertex v_0, our goal is to find all the vertices that are reachable from v_0. We can view this problem as searching for a subset of vertices of $W \subseteq V$ such that $v_0 \in W$ and $\forall x, y : (x \in W) \wedge (x, y) \in E$ implies that $y \in W$. We model W using a boolean array G such that $v_i \in W$ iff $G[i]$ is true.

The following LLP algorithm computes the least G (or the smallest W) that satisfies

$$B_{traverse} \equiv G[0] \wedge (\forall (v_i, v_j) \in E : G[j] \geq G[i])$$

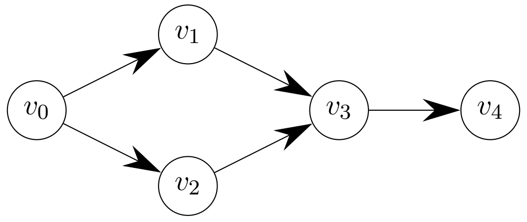

Figure 8.1: A directed graph

The predicate B requires v_0 to be reachable from v_0 and for all edges (v_i, v_j) if v_i is reachable then so is v_j. It can be verified that $B_{traverse}$ is a lattice-linear predicate. It is a conjunctive predicate and the first conjunct is a local predicate. We only need to show that the second conjunct is also lattice-linear. If the second conjunct is false, then there exists $(v_i, v_j) \in E$ such that $G[i]$ equals 1 and $G[j]$ equals 0. In this case, unless $G[j]$ is also set to 1, the predicate can never become true. Therefore, we get the following LLP algorithm.

Algorithm LLP-Traversal: Finding the reachable set of vertices.

```
Process $P_j$
input: $edges(j)$: list of $1..n$;
init($j$): if $(j = 0)$ $G[j] := 1$ else $G[j] := 0$;
forbidden($j$): $\exists i : j \in edges(i) : (G[i] = 1) \wedge (G[j] = 0)$
    advance($j$): $G[j] := 1$;
```

LLP Algorithm is non-deterministic and one could use different strategies to compute forbidden indices. For example, one can maintain a set S that contains *frontier* vertices – set of vertices that are reachable but their outgoing edges have not been explored yet. Algorithm Traversal-Sequential gives a sequential implementation of the LLP algorithm with S as the set of frontier vertices. By using different strategies for removing an index from S, we can traverse the graph in different order. If S is maintained as a queue, the removal corresponds to removing from the head of the queue, and the addition corresponds to appending at the end of the queue, we traverse the graph in a *breadth-first* manner. If S is maintained as a stack such that removal corresponds to *pop*() and the addition corresponds to *push*() method on the stack, then we traverse the graph in a depth-first search manner.

Algorithm Traversal-Sequential: Finding the reachable set of vertices from v_0 in a Directed Graph

```
var $G$: vector of int initially $\forall i : G[i] = 0$;
    $S$: set of indices;
$S.add(0)$; // add 0 as the initial forbidden index
$G[0] := 1$;
while $\neg S.empty()$ do
    $j := S.removeAny(G)$; // remove any index from $S$
    forall $(k \in dep(j))$
        if $(G[k] = 0)$ // $k$ is forbidden
            $G[k] := 1$; // advance on $k$
            add $k$ to $S$;
endwhile;
return $G$; // the optimal solution
```

8.3 Breadth First Search Tree in a Directed Graph

In the previous section, we saw an algorithm to compute the set of reachable vertices from a given vertex. Now suppose that we are interested in computing the breadth-first-search tree rooted at any given node v_0. We first define the search lattice L to be the set of distance vectors where $G[i]$ is the distance of vertex i from v_0. Initially, $G[0]$ is zero for the vertex v_0 and ∞ for all other vertices. We define G to be feasible if

$$B_{bfs}(G) \equiv \forall j \neq 0 : \forall (i, j) \in E : G[j] \leq G[i] + 1.$$

Our goal is to maximize G subject to B_{bfs}.

A vertex j is defined to be forbidden if it has a predecessor i such that $G[j] > G[i] + 1$. If none of the vertices is forbidden, we get that $\forall j \neq 0 : \forall (i,j) \in E : G[j] \leq G[i] + 1$. Whenever a vertex j is forbidden, we advance the index j by setting $G[j]$ to $G[i] + 1$.

Algorithm LLP-Traversal-BFS: Finding the reachable set of vertices using BFS.

```
Process $P_j$
input: $pre(j)$: list of $1..n$;
init($j$): if $(j = 0)$ $G[j] := 0$ else $G[j] := \infty$;
forbidden($j$): $\exists i \in pre(j) : G[j] > G[i] + 1$
    advance($j$): $G[j] := G[i] + 1$
```

The algorithm terminates when there is no forbidden vertex. Any vertex k such that $G[k]$ is infinite is not reachable from the vertex v_0.

Algorithm LLP-Traversal-BFS is based on using the predecessor information for each vertex. Alternatively, we can use the adjacency list for each vertex. We now show an implementation of breadth-first-search, when we are also required to maintain a variable *parent* that gives the parent of any node x in the graph. If x is reachable from v_0, then by following the *parent* pointer from x to v_0, we will get a shortest path from v_0 to x. The set S keeps the set of reachable indices. Initially, only v_0 is the reachable vertex. We explore all the outgoing edges from any vertex $v_j \in S$. The set T stores all the vertices that become reachable when vertices in S are explored. If any vertex k adjacent to v_j gets its $G[k]$ reduced because of the edge (v_j, v_k), it is added to the set T. This exploration is continued until there are no unexplored vertices.

Algorithm BFS-Traversal-Parallel: Parallel Algorithm BFS to find the Breadth-First-Search Tree in a Directed Graph

```
var $G$: vector of int initially $\forall i \neq 0 : G[i] = maxint; G[0] := 0$;
    $parent$: vector of int initially $\forall i : parent[i] = -1$; // null parent
    $S$: set of indices initially $\{0\}$; // add $v_0$ as the initial forbidden index
while $\neg S.empty()$ do
    $T$ := set of indices initially $\{\}$;
    forall $(j \in S) \wedge (k \in dep(j))$ in parallel do:
        if $(G[k] > G[j] + 1)$
            $G[k] := G[j] + 1$;
            $parent[k] := j$;
            add $k$ to $T$;
    endforall;
    $S := T$;
endwhile;
return $G$;
```

8.4 Depth-First Search Tree in a Directed Graph

Another way to traverse a graph is the depth-first search method. While the breadth-first search traverses the graph one layer at a time, the depth-first search continues to take a path as far as it can take and then it backtracks to traverse the remaining graph (in the same fashion!).

Algorithm DFS-Traversal: Sequential Algorithm DFS to find the Depth-First-Search Tree in a Directed Graph

```
var $G$: vector of int initially $\forall i : G[i] = 0$;
    $parent$: vector of int initially $\forall i : parent[i] = -1$; // null parent
    $S$: stack of indices initially empty;
    $discovered$: vector of int;
    $finished$: vector of int;
    $tick$: integer initially 1;
$S.add(s)$; // add $s$ as initial forbidden index
while $\neg S.empty()$ do
    $j := S.pop()$; // remove an index from $S$
    if $(G[j] = 0)$ then
        $G[j] := 1$;
        $discovered[j] := tick$;
        $tick := tick + 1$;
        forall $(k \in dep(j)) \wedge (G[k] = 0)$:
            $parent[k] := j$;
            $S.push(k)$;
        endforall;
        $finished[j] := tick$;
        $tick := tick + 1$;
    endif;
endwhile;
return $G$;
```

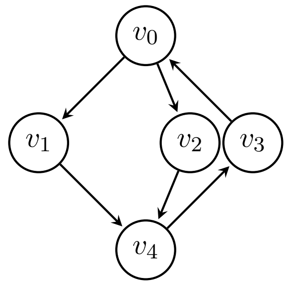

Figure 8.2: An Example of DFS-Traversal

Node	discovered	finished
v_0	1	10
v_1	2	7
v_4	3	6
v_3	4	5
v_2	8	9

Figure 8.3: The values of *discovered* and $finished$ for vertices with DFS on the graph in Fig. 8.2

Algorithm DFS-Traversal visits all the vertices reachable from the vertex s in a depth-first search manner. The variable G is used the mark all the vertices that are reachable from s. The variable S is used as a stack to store the vertices while traversing the graph. The variable *discovered* is used to store the time (tick) when each vertex in the graph is explored. The variable $finished$ is used to store the tick when the algorithm is finished exploring the vertex. The variable *parent* is used to store the depth-first-tree. Anytime a vertex k is visited for the first time from a vertex j, the $parent[k]$ is recorded as j.

The variables *discovered* and $finished$ are used only for recording the *tick* when a vertex is started to be explored and finished for exploration.

Fig. 8.2 shows the DFS traversal on a graph. Suppose we start the DFS traversal from the node v_0. Then, node v_1 is visited next. After v_1, we visit the node v_4. From node v_4, we visit the node v_3. When v_3 is explored, we get to the node v_0 which has already been visited. We backtrack to node v_4 which does not have any other outgoing edge. From v_4, we backtrack to v_1 and then to v_0. Now, from v_0, we can visit the node v_2. From v_2, we can go to node v_4 which has already been visited. Hence, we backtrack to node v_0. At this point, all the outgoing edges from v_0 have been traversed and the DFS traversal is done. The variables for DFS traversal for this graph are shown in Fig. 8.3.

8.5 Topological Sort of a Directed Acyclic Graph

Suppose that we are given a directed graph with no cycles. Such a graph can be "layered" as follows. Every vertex i in the graph is assigned a number $G[i]$ such that if there is a directed edge from i to j, then the number assigned to i is strictly less than that of j, i.e., for all edges $(i,j) \in E$, $G[i]$ is less than $G[j]$. An application of this concept is the prerequisite structure of courses in a University and the integer assigned to the course corresponds to the earliest semester in which the course can be taken by a student.

In this application, our lattice consists of vectors of natural numbers. We are determining the least vector, G, that satisfies the predicate

$$B_{layer} \equiv \forall (i,j) \in E : G[i] < G[j].$$

The predicate is lattice-linear because if it is false then there exist $(i,j) \in E$ such that $G[i] \geq G[j]$. In this case, the index j is forbidden and unless $G[j]$ is advanced B_{layer} can never become true. For efficiency, we define a predicate

$$fixed(j) \equiv \forall (i,j) \in E : G[i] < G[j].$$

Now, the predicate B_{layer} can be rewritten as

$$B_{layer} \equiv \forall j \in [n] : fixed(j).$$

We will use $fixed[j]$ as a variable in our program. Algorithm shown in Fig. LLP-Layering gives the layering of a directed acyclic graph. We use the boolean array $fixed$ to mark the vertices whose level number in G are final. Initially, all vertices that do not have any predecessors are $fixed$ and labeled with 0. A vertex is forbidden if it is not fixed and all its predecessors are fixed. Whenever a vertex j is forbidden, we mark its level as 1 more than any of its predecessors. In this algorithm, $G[j]$ gives the length of the longest path from any initial vertex to j.

Algorithm LLP-Layering: Layering of a Directed Acyclic Graph.

```
input: $pre(j)$: list of $1..n$;
init(j): $G[j] := 0$;
    if $pre(j) = \{\}$ then $fixed[j] := true$ else $fixed[j] := false$;
forbidden(j): $\neg fixed[j] \wedge \forall i \in pre(j) : fixed[i]$
    advance(j): $G[j] := \max_{i \in pre(j)} G[i] + 1$; $fixed[j] := true$;
```

The algorithm takes parallel time equal to the length of the longest path in the directed acyclic graph. Its work complexity is $O(n+m)$.

In many applications, we are interested in coming up with a *total order* on all vertices such that for all edges (i,j), $G[i]$ is strictly less than $G[j]$. This operation is called *topological sort* of a directed acyclic graph. The reader is invited to modify LLP-Layering for such applications.

8.6 Connected Components in an Undirected Graph: Slow Algorithm

We are given an undirected graph (V,E) on n vertices numbered $1 \ldots n$. Our goal is to label each vertex i with $G[i]$ such that $G[i]$ is the largest numbered vertex in the component to which i belongs. A vector G is defined to be feasible if

$$(\forall i : G[i] \geq i) \wedge (\forall (i,j) \in E : G[i] \geq G[j])$$

We consider the lattice of n dimensional vectors with natural $\leq$ relation on the vectors. We first claim that

Lemma 8.1 *$B_{comp}(G) \equiv (\forall i : G[i] \geq i) \wedge (\forall (i,j) \in E : G[i] \geq G[j])$ is a lattice linear predicate.*

Proof: We show that each of the conjuncts is lattice linear. The first conjunct $(\forall i : G[i] \geq i)$ is lattice linear because it is a conjunction of local predicates. The second conjunct $(\forall (i,j) \in E : G[i] \geq G[j])$ is lattice linear because if there exists $(i,j) \in E$ such that $G[i] < G[j]$, then unless $G[i]$ is advanced to $G[j]$, the predicate B_{comp} can never become true.

■

Lemma 8.2 *The least vector G in the lattice that satisfies B_{comp} gives us the labeling such that $G[i]$ is the largest numbered vertex in the component to which i belongs.*

Proof: Use induction on the distance of i from the largest numbered vertex in its component.

■

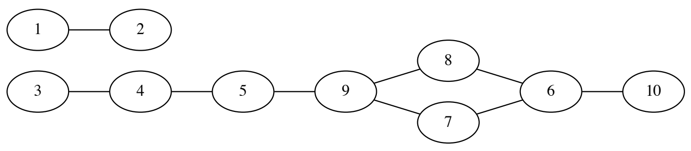

Figure 8.4: An undirected graph

Algorithm SlowComponents: Algorithm *SlowComponent* to find all the connected components of an undirected graph.

```
1    input: $adj(j)$: list of $1..n$;//adjacency list of vertex $j$
2    init: $G[j] := j$;
3    ensure: $G[j] \geq \max\{G[i] \mid i \in adj(j)\}$
```

By applying Lattice-Linear Predicate method, we get the Algorithm SlowComponents. We use n threads in the algorithm, one for each vertex j, responsible to update $G[j]$. Every thread j initializes $G[j]$ to j. If any thread j finds that there exists i such that i and j are adjacent and $G[j]$ is less than $G[i]$, then it updates its component to $G[i]$.

First, it is easy to verify that the algorithm *SlowComponents* maintains the invariant that $G[i] = j$ implies that i and j are in the same connected components. Initially, the invariant holds because $G[j]$ is set to j. Any update to $G[j]$ is to some $G[i]$ such that i is adjacent to j. By induction, $G[i]$ is in the same connected component as i and because i is connected to j, the invariant continues to hold for $G[j]$. Conversely, because LLP algorithm returns the least vector that satisfies B_{comp}, we get that the algorithm terminates with $G[i]$ as the largest numbered vertex in the component to which i belongs on account of Lemma 8.2.

Let us analyze the time complexity of the above algorithm in a synchronous model in which every step consists of a thread j determining whether it is forbidden, and then advancing to the maximum $G[i]$ such that $(i, j) \in E$. The algorithm may require as many as $O(n)$ steps in the worst case when nodes are arranged in a long chain with the largest node at one end of the chain.

8.7 Connected Components: A Faster Algorithm

We now show the application of *pointer jumping* technique to this problem. We show an LLP algorithm that uses pointer jumping. In this algorithm, we use the notion of prioritized forbidden function. In many situations, an index may be forbidden due to multiple reasons. Determining whether an index is forbidden requires some work and it makes sense to first compute the forbidden functions that are easy to compute. The forbidden function in Algorithm SlowComponents requires a node to scan all its adjacent neighbors. If we view $G[i]$ as the parent of i, then it is clear that $G[i]$ should also be greater than or equal to $G[G[i]]$. Checking violation of this condition is much simpler that scanning all adjacent neighbors. This step can be formalized as **ensure** $G[j] \geq G[G[j]]$.

Instead of using G as our state vector we use more user-friendly name *parent* as the state vector. Then, the above step can be implemented as follows:

```
while $(\exists j : parent[j] < parent[parent[j]])$
    for all $j \in V$ in parallel do
            $parent[j] := parent[parent[j]]$
```

If the parent pointers form a rooted tree, then the pointer jumping technique reduces the height of the tree by making every node point to its grandparent in one time step. In $O(\log n)$ iterations of the while loop every *parent* pointer would be equal to the *grandparent* pointer.

Combining the predicate corresponding to pointer-jumping with the predicate that updates the parent pointer with the largest parent pointer in the neighborhood of any node, we get the algorithm LLP-FastComponents.

Algorithm LLP-FastComponents: Finding all the connected components of an undirected graph.

```
1  input: $adj(j)$: list of $1..n$;//adjacency list of vertex $j$
2  init: $parent[j] := j$;
3  ensure(1): $\forall i : parent[i] \geq parent[parent[i]]$
4  ensure(2): $\forall i : parent[i] \geq \max\{parent[j] \mid (\forall(i,j) \in E\}$
```

The algorithm has two *ensure* statements. The first ensure statement guarantees that node i has its parent equal to its grandparent. The second ensure statement guarantees that node i has its parent pointer at least as big as its neighbors. This statement alone would have been enough to find the largest label in a component. However, adding the first ensure statement speeds up the process.

Our implementation of the algorithm will execute the first ensure statement until it is globally true. In a synchronous model, this step will take $O(\log n)$ parallel time. When the first ensure statement is met, the directed graph created from *parent* pointer can be viewed as a collection of rooted stars. Every rooted star is part of the same components. However, nodes in the same components may belong to different rooted stars. The second ensure statement merges these stars. If there is no edge across rooted stars, then we are done and every rooted star is a component by itself. Otherwise, every rooted star computes the largest parent pointer known to any node in the rooted star and point to it. We can view each rooted star as a *supervertex*. Two vertices are in the same supervertex if their parents are the same. Once we have computed the largest parent known to a supervertex, if the new parent pointer discovered is bigger than the earlier root, then this tree merges to another tree. Otherwise, this rooted tree does not join any other rooted tree. We show that the number of rooted stars reduce by a factor of two in every two such steps. If there is any edge from this rooted star i to another rooted star j, then there are three cases:

1. If $parent[i]$ is less than $parent[j]$, then the star i joins the star j.
2. If $parent[j]$ is less than $parent[i]$ and $parent[i]$ is the largest parent pointer known to star j, then star j joins star i.
3. If $parent[j]$ is less than $parent[i]$ but $parent[i]$ is not the largest parent pointer known to star j, then star j joins some other star k that has bigger parent pointer than star i. In this case, we know that whenever we scan parents of the neighbors next time star i will discover a bigger star and join that star.

From the above analysis, we get that the number of rooted stars go down by a factor of 2 in at most two steps of scanning neighbors of supervertices.

Whenever, we merge rooted stars, we get rooted trees and we can apply the pointer-jumping again. The algorithm terminates when no rooted star can be merged with any other rooted star. At that point, every rooted star corresponds to a component of the graph and the required labeling is given by the parent pointer.

Algorithm FastComponents gives the entire algorithm. It initializes parent node of every node i as itself (line 4). Thus, initially every node is a rooted star by itself. It has a *while* loop that checks for the forbidden condition that there exists an edge between two nodes with different *parent* (line 5) If there is no such edge, then we are done and every rooted star corresponds to a component. Otherwise, every node computes the largest parent that it or one of its neighbor knows (line 8). Once, every vertex has computed this value, we merge the rooted stars by making the root point to the largest parent pointer known by the supervertex. This operation will merge rooted star to the biggest rooted star know to it and create rooted trees. We can then do pointer jumping to create new rooted stars and go back to the while loop.

It consists of $O(\log n)$ iterations such that each iteration takes $O(\log n)$ time giving us the time complexity of $O(\log^2 n)$.

Algorithm FastComponents: Finding Connected Components

```
Input: Undirected Graph: $(V, E)$.
Output: Labeling of vertices by the component id
var
    $parent$: array[1..$n$] of 0..$n$ initially all $\forall i : parent[i] = i$;
while $(\exists (v,w) \in E : parent[v] < parent[w])$
    // Step 1: Compute $vmax$ for every vertex
    forall $v \in V$ in parallel do
        $vmax[v] := \max\{parent[w] \mid$ such that $(v,w) \in E \vee (w = v)\}$
    // Step 2: Make the $parent$ of $v$ as the largest $vmax$ in its neighborhood
    forall $v \in V$ in parallel do
        if $v = parent[v]$ then //$v$ is the root of the star
            $parent[v] = \max\{vmax[u] | parent[v] = parent[u]\}$
    // Step 3: Convert every rooted tree into a rooted star
        while $(\exists j : parent[j] \neq parent[parent[j]])$
            for all $v \in V$ in parallel do
                $parent[v] := parent[parent[v]]$
endwhile
```

We now give an example of the execution of the algorithm on the graph shown in Fig. 8.4. It is easy to see visually that there are two connected components in this graph.

8.8 Summary

The following table lists all the algorithms discussed in this chapter.

Problem	**Algorithm**	**Parallel Time**	**Work**
Breadth-First Search	BFS-Traversal-Parallel	critical path length	$O(m+n)$
Topological Sort	LLP	critical path length	$O(m+n)$
Components	Slow LLP	$O(n)$	$O(m+n)$
Components	Fast LLP	$O(\log^2 n)$	$O(m+n)$

8.9 Problems

1. Algorithm LLP-Layering gives a level number $G[i]$ for each vertex i such that for any edge (i,j), we have that $G[i]$ is strictly less than $G[j]$. Modify the algorithm to generate a total order on all vertices.

2. Algorithm LLP-Layering gives us the least integral labels satisfying that for all edges (i,j), $G[i]$ is strictly less than $G[j]$. Such a property is possible only for acyclic graphs. Modify the algorithm to output "error" whenever there is a cycle in the input graph.

8.10 Bibliographic Remarks

The reader is referred to [CLRS01] for basic graph algorithms.

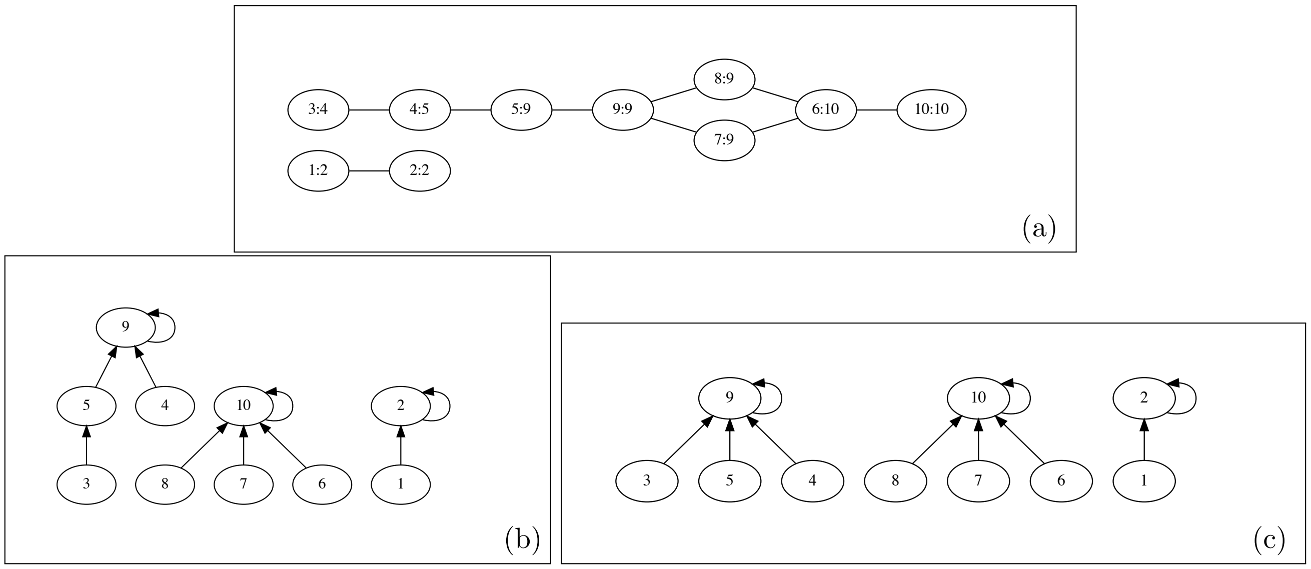

Figure 8.5: (a) *vmax* for every node in Fig. 8.4 . (b) Pointing to largest *vmax* to get Rooted-trees (c) Rooted-stars at the end of first iteration.