**Algorithm 1:** Function Main($G$, $PES$)

**Input**: An instance of the $A$-$CONN$ problem where $G$ has a partial $k$-tree topology with a given $PES$=$(v_1, v_2, \ldots, v_n)$
**Output**: $Conn(G)$
**Notation:** $Temp$ is a temporary table.

1 **Initialization:** initialize a table $T_H$ for each $k$-clique $H$ of $G$.
  $T_H$ contains all possible state types on nodes of $H$.
2 **for** $(i = 1, 2, \ldots, |V| - k)$ **do**
3   $Temp = T_{v_i,1}$
4   **for** $(j = 2, 3, \ldots, k)$ **do**
5     $Temp = t\_merge(Temp, T_{v_i,j})$
  **end**
6   $T_{v_i,base} = t\_merge(Temp, T_{v_i,base})$
7   **foreach** $(key \in T_{v_i,base})$ **do**
8     **if** ($v_i$ *is a singleton part of key*) **then**
9       delete $key$ from $T_{v_i,base}$
    **end**
    **else**
10       delete $v_i$ and its associated position from $key$
    **end**
  **end**
**end**
11 **return** $Conn(G) = \sum$ values in table $T_{v_{n-k},base}$ corresponding to state types that have exactly one connected component

$K_{v_i,base}$ is considered bad at this stage if node $v_i$ appears as a singleton part in the partition specified by $A$-$type(S)$.

This badness holds because $K_{v_i,base}$ is a separator clique in $G$. So, if $v_i$ appears disconnected from the nodes in $K_{v_i,base}$ in any network state $S$ then $S$ can not be extended to an operating state of the entire network. On the other hand, if $A$-$type(S)$ is not bad, then step 10 just removes node $v_i$ and its associated position from $A$-$type(S)$.

**Example 4.4.** Figure 4.3 illustrates a 3-tree that has a $PES = (v_6, v_5, v_7, v_8, v_9, v_4)$. the main loop processes node $v_6$ by merging tables $T_{v_6,1}$ on nodes $\{v_4, v_5, v_6\}$, $T_{v_6,2}$ on nodes $\{v_3, v_5, v_6\}$, $T_{v_6,3}$ on nodes $\{v_3, v_4, v_6\}$, and $T_{v_6,base}$ on nodes $\{v_3, v_4, v_5\}$ into one table stored in $T_{v_i,base}$. ■

**Termination (Step 11):** The main loop finishes after processing node $v_{n-k}$. The resulting table $T_{v_{n-k},base}$ corresponds to the $k$-clique $H$ on nodes $(v_{n-k+1}, \ldots, v_n)$.

Any state type where all nodes of $H$ appear in one connected component corresponds to a set of operating network states of the entire network . Thus, we return the sum of probabilities of all such state types.

### 4.2.2 Function Table Merge

**Algorithm 2:** Function $t\_merge(T_1, T_2)$

**Input**: Two tables $T_1$ and $T_2$ that may share common nodes
**Output**: A merged table $T_{out}$

**Initialization:** Clear table $T_{out}$
**set** $C$ = the set of common nodes between $T_1$ and $T_2$
**foreach** (*pair of state types* $key_1 \in T_1$ *and* $key_2 \in T_2$) **do**
  **if** (*any node in* $C$ *lies in two different positions in* $key_1$ *and* $key_2$) **then**
    **continue**
  **end**
  **set** $key_{out}$ = the state type obtained from node positions in $key_1$ and $key_2$, and the partition computed by $p\_merge(key_1, key_2)$
  **set** $p_{out} = T_1(key_1) \times T_2(key_2)$ adjusted to take the effect of common nodes in $C$ into consideration
  **if** ($key_{out} \in T_{out}$) **then**
    update $T_{out}(k_{out})$ +=$p_{out}$
  **end**
  **else**
    **set** $T_{out}(key_{out}) = p_{out}$
  **end**
**end**
**return** $T_{out}$

Algorithm 2 illustrates the main steps of the middle level table merge function. We recall that $t\_merge$ is called from the main function to merge two tables, denoted $T_1$ and $T_2$, that may have a set $C$ of common nodes (i.e., $V(T_1) \cap V(T_2) \neq \emptyset$). Each table $T_i, i = 1, 2$, stores information about some subgraph, $G_i$, that has been reduced onto some $k$-clique, $K_i$. Table merging is done by processing each pair of state types (i.e., table keys) $key_1 \in T_1$ and $key_2 \in T_2$. Processing state types $key_1$ and $key_2$ results in a new state type, denoted $key_{out}$, and an associated probability, denoted $p_{out}$.

Our method of computing $P_{out}$ can be summarized as follows. Each partition (e.g. $P_1$ or $P_2$) is represented by a list of sets (a list is an ordered container). Each part within a partition is represented as a set of nodes.

First, we put all parts of $P_1$ and $P_2$ in one partition. In function *p_merge* (Algorithm 3) we choose to put all parts in $P_1$ (steps 1 and 2). Second, we process each pair of parts in $P_1$ by merging them into one part if there is at least one common node. If two parts are merged together, then both parts are deleted from the ordered list $P_1$ and their union is placed at the beginning of $P_1$. Processing of pairs of parts in the updated $P_1$ then restarts from the beginning of $P_1$. Processing finishes when all pairs in the ordered list $P_1$ are considered.

**Algorithm 3:** function *p_merge*($P_1$, $P_2$)

**Input**: Two partitions $P_1$ and $P_2$

**Output**: A partition $P$

**Notation:** $s$ and $t$ are two set iterators and their corresponding set are indicated by $s^*$ and $t^*$.

```
1  foreach ( set s* in P2) do
2      P1.push_back(s*)
   end
3  for (s = P1.begin(); s ≠ P1.end(); ++s) do
4      for (t = s.next(); t ≠ P1.end(); ++t) do
5          if (s* ∩ t* ≠ ∅) then
6              P1.push_front(s* ∪ t*)
7              P1.delete(s*)
8              P1.delete(t*)
9              s = P1.begin()
10             break
           end
       end
   end
11 set P = P1
   return P
```

## 4.3 Example Tables

**Example 4.6.** In figure 4.5, two keys (state types) $key_1 = (\{v_1, v_2\}^{(1,1)}\{v_3\}^{(2)})$ and $key_2 = (\{v_1\}^{(1)}\{v_3, v_4\}^{(2,1)})$ belong to tables $T_1$ and $T_2$ respectively. The two keys can be merged together since the common nodes $v_1$ and $v_3$ assume the