APPENDIX: SUMMARY OF MAIN NOTATION

G	An arbitrary graph.
V	The set of nodes in G.
E	The set of edges in G.
S	A coalition, i.e., a subset of nodes.
2^V	The set of all subsets of nodes.
$E(S)$	The set of all edges between members of S.
$G(S)$	The subgraph of G that is induced by S, i.e., $G(S) = (S, E(S))$.
$\mathcal{K}(S)$	The partition of S whose subsets induce the connected components in $G(S)$.
v_i	A node in the graph.
$N(v_i)$	The set of neighbors of node v_i.
$M(v_i)$	The set of neighbors of node v_i sorted descendingly based on their degree.
$N(S)$	The set of neighbors of coalition S, i.e., $N(S) = \bigcup_{v_i \in S} N(v_i) \setminus S$.
$\mathcal{C}$	The set of all connected induced subgraphs of G.
f	The characteristic function, which specifies the value of every coalition.
f_G	The function that specifies the value of every connected coalition of G.
$SV_i(f)$	The Shapley value of node v_i.
$MV_i(f_G)$	The Myerson value of node v_i.
$f_G^{\mathcal{M}}$	The characteristic function of a Myerson game ($\mathcal{M}$ stands for Myerson).
$f_G^{\mathcal{L}}$	The characteristic function of a connectivity game ($\mathcal{L}$ stands for Lindelauf et al.).
$f_G^{\mathcal{A}}$	The characteristic function of a 0-1-connectivity game ($\mathcal{A}$ stands for Amer and Giménez).
π	A permutation of nodes from V.
Π	The set of all permutations of V.
S_i^π	The coalition consisting of v_i and all the players that precede v_i in permutation π.
ω_i	The weight of the node $v_i \in V$.
ω_{ij}	The weight of the edge $\{v_i, v_j\} \in E$.

APPENDIX: DFSE ALGORITHM—AN ILLUSTRATION

Figure 4 provides a graphical overview of the workings of DFSE, by illustrating how the state of (S, *Forbidden*) is modified during the execution of the recursive function *ExpandSubgraph*.

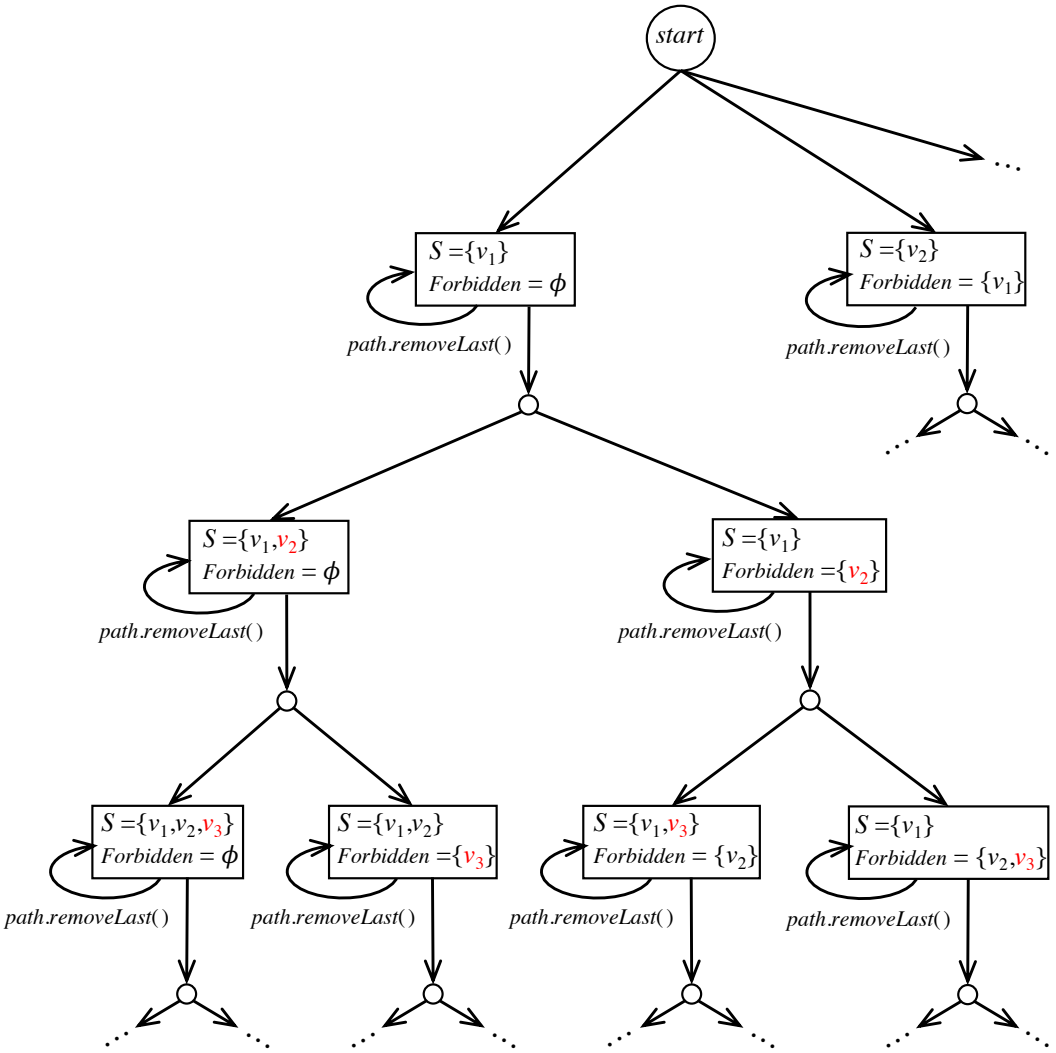


Fig. 4. A graphical overview of the workings of DFSE, which illustrates how the state of (S, *Forbidden*) is changed during the execution of the recursive function *ExpandSubgraph*. Here, every rectangle represents a certain state of (S, *Forbidden*). For every such state, the process can be divided into two processes: the first adds a node to S and to *path* (in line 13), whereas the second adds that node to *Forbidden* (in line 14) to ensure that it is never added again to S; this division is illustrated by the vertical arrow pointing downwards, which then splits into two. The only exception is when lines 13 and 14 are not executed, in which case the last node in *path* is removed (in line 15), after which either S is printed (in line 21), or *ExpandSubgraph* is called with the same instance of (S, *Forbidden*) but with the new *path* (in line 19); this is illustrated by the rounded arrow, which returns to the same rectangle because S and *Forbidden* remain unchanged.

Figure 5 depicts a sample graph, G, and illustrates how DFSE generates each of the subgraphs containing v_1. In each subfigure, the white nodes are those in the subgraph S, whereas the black nodes are those in *Forbidden*. Here, arrows represent the moves made by DFSE, which are either an *expansion move* (represented by an arrow pointing downward) or a *backtracking move* (represented by an arrow pointing upward). The algorithm starts by setting $S = \{v_1\}$ (in line 6), and then makes the first call to the function *ExpandSubgraph* (in line 7), which is responsible for generating all subgraphs containing v_1. In more detail, this function generates all subgraphs depicted in Figure 5, in their respective order. As a concrete example, let us describe how the first such subgraph is generated:

- The arrow labeled 1 represents the 1^{st} move, which expands S by adding to it the first neighbor of v_1, namely v_2 (this is carried out in lines 11–13, by setting $u = v_2$ and then making a recursive call with $S \cup \{u\}$);
- In a similar manner, the arrows labeled 2 and 3 represent the 2^{nd} and 3^{rd} moves, whereby S is expanded by adding to it v_4 and v_3, respectively. At this point, $path = (v_1, v_2, v_4, v_3)$ and $S = \{v_1, v_2, v_3, v_4\}$;
- Since there are no neighbors of v_3 that can be added to S, we reach line 15, where the algorithm starts backtracking. This is done in the 4^{th} move, whereby the last node in $path$ is removed (the removal is carried out in lines 15–19);
- Likewise, since there are no neighbors of v_4 that can be added to S, the 5^{th} move backtracks to v_2;
- The 6^{th} move expands S by adding to it v_5. At this point, $path = (v_1, v_2, v_5)$ and $S = \{v_1, v_2, v_3, v_4, v_5\}$;
- Since there are no neighbors of v_5 to be added to S, the algorithm backtracks to v_2 in the 7^{th} move;
- Following the same reasoning, the algorithm backtracks to v_1 in the 8^{th} move. At this point, $path = (v_1)$ and $S = \{v_1, v_2, v_3, v_4, v_5\}$;
- Finally, since there are no neighbors of v_1 that can be added to S, in line 15, $path$ becomes empty and the algorithm outputs $S = \{v_1, v_2, v_3, v_4, v_5\}$.

In this example, every move in which the algorithm expands the subgraph—namely, the 1^{st}, the 2^{nd}, the 3^{rd} and the 6^{th} move—is made via a recursive call to *ExpandSubgraph*, whereby a node, u, is added to S (line 13). Each such call ultimately enumerates all connected induced subgraphs whose nodes form a superset of $S \cup \{u\}$. After that, the algorithm reaches line 14, where the node u is added to the list *Forbidden*, to ensure that none of the subsequently-enumerated subgraphs contains u. For instance, the 2^{nd} move adds v_2 to S via a recursive call to *ExpandSubgraph*; this call ultimately enumerates all subgraphs containing $\{v_1, v_2\}$, i.e., those depicted in the subfigures (a) to (h) of Figure 5. After that, v_2 is added to *Forbidden*, to ensure that none of the subsequently-generated subgraphs contain v_2 (see how v_2 is highlighted in black in subfigures (i) to (n) of Figure 5).

For comparison purposes, Figure 6 illustrates how BFSE generates each of the subgraphs containing v_1 for the same graph G. In each subgraph, the white nodes are those in the set $Old \cup New$, whereas the black nodes are those in *Forbidden*, but not in $Old \cup New$. Arrows represent the moves made by BFSE. In each such move the whole set of neighbors of nodes from the already-found subgraph is considered and a different subset of those nodes is added to a subgraph. In particular, graphs (b), (d), (f), (g), (i), (k), (m) correspond to 7 different non-empty subsets of neighbors of v_1, i.e., $\{v_2, v_3, v_5\}$. For each such a subset further neighbors are considered. For 6 cases new neighbor—v_4—is found. For case (f) (i.e., subset $\{v_5\}$) no new nodes can be reached.

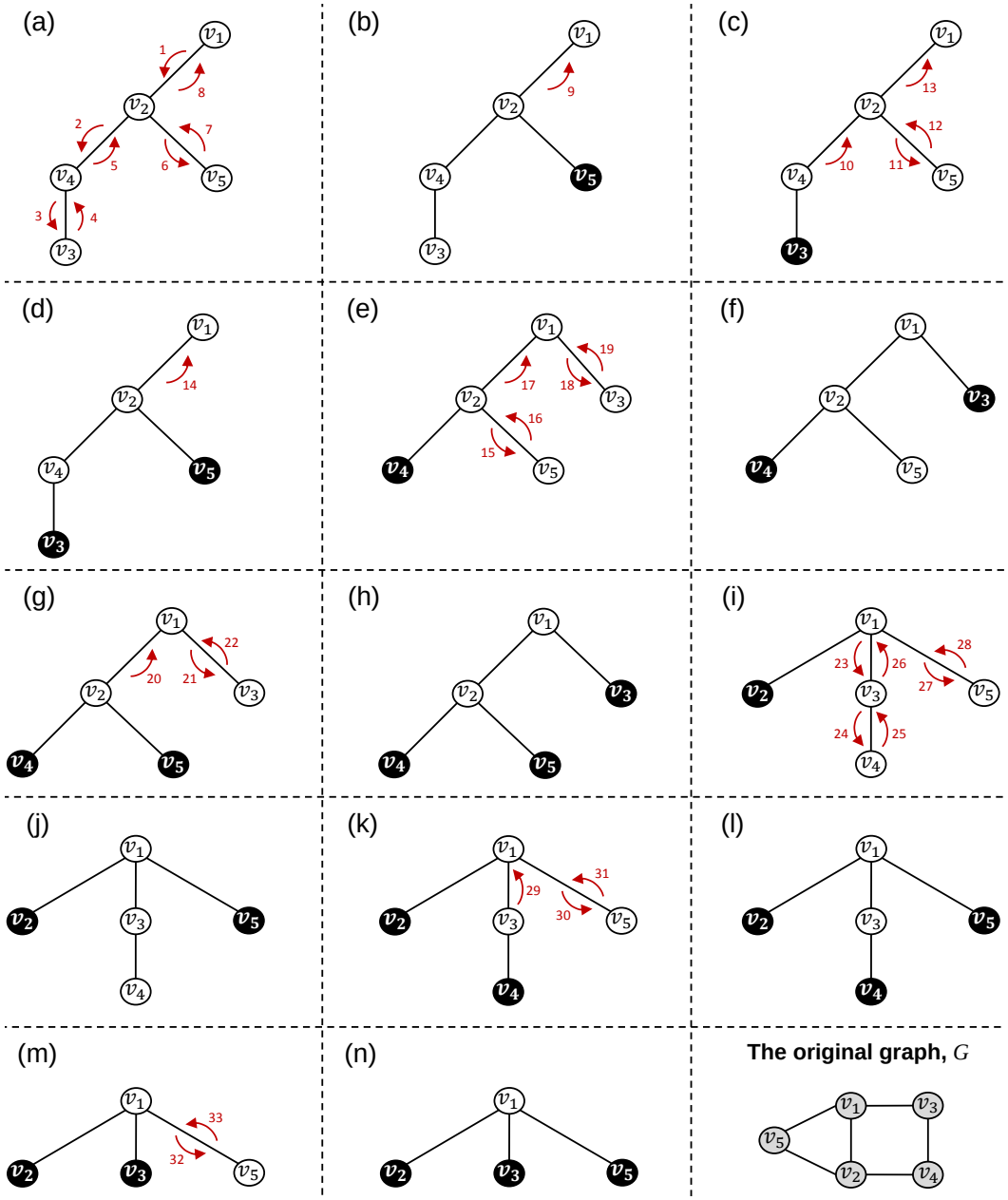


Fig. 5. A sample graph, G, along with an illustration of how DFSE generates the subgraphs that contain v_1 (the order of the subgraphs follows the order in which DFSE generates those subgraphs). In each subfigure, the white nodes are those in the subgraph, S, whereas the black nodes are those in *Forbidden*. Here, arrows represent the moves made by DFSE; these are either an *expansion move* (represented by an arrow pointing downward) or a *backtracking move* (represented by an arrow pointing upward).

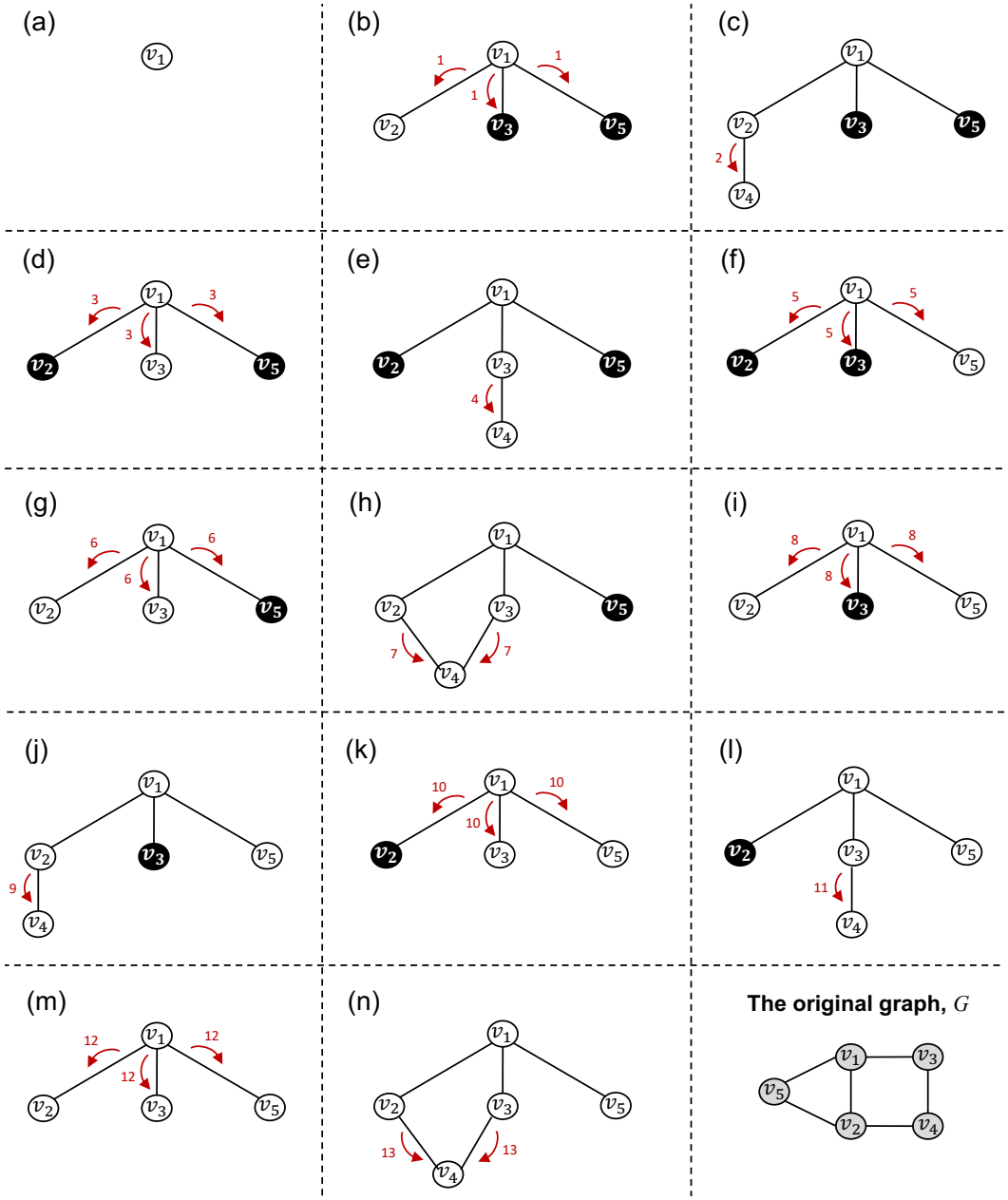


Fig. 6. A sample graph, G, along with an illustration of how BFSE generates the subgraphs that contain v_1 (the order of the subgraphs follows the order in which BFSE generates those subgraphs). In each subfigure, the white nodes are those in the subgraph, $Old \cup New$, whereas the black nodes are those in *Forbidden*, but not in $Old \cup New$. Here, arrows represent the moves made by BFSE.

APPENDIX: PROOFS

Proof of Lemma 4.1. It is easily verifiable that the above conditions are satisfied every time *ExpandSubgraph* is called in line 7. It remains to show that if those conditions are satisfied in the header of the function (i.e., satisfied in line 8), then they are also satisfied in the recursive calls of the function (i.e., in lines 13 and 19). To this end, we will consider each condition separately.

(a) Assume that $S \cap Forbidden = \emptyset$ in line 8. Then, all we need to show is that this equality holds in the next recursive call to *ExpandSubgraph*. This call can be made in either line 13 or line 19. As for the call in line 13, the condition in line 12 ensures that $(S \cup \{u\}) \cap Forbidden = \emptyset$. As for the call in line 19, the set S remains unchanged compared to its initial state in line 8, unlike the set *Forbidden* which may have been modified by adding to it (one or more) nodes in line 14. However, each of these nodes is surely not in S; this is guaranteed by the condition in line 12. Consequently, in line 19 we have: $S \cap Forbidden = \emptyset$.

(b) Assume that in line 8 we have: $path \subseteq S$ and that the nodes in the parameter *path* actually do form a path in G. It remains to show that the same holds in the next recursive call to *ExpandSubgraph* (either in line 13 or 19). As for the call in line 13, since $path \subseteq S$, then clearly: $path \cup \{u\} \subseteq S \cup \{u\}$. Moreover, since the nodes in *path* form a path in G, and since u is a neighbor of the last node in *path*, then the nodes in $path \cup \{u\}$ also form a path in G. As for the call in line 19, the set S remains unchanged compared to its initial state in line 8, whereas *path* is modified by removing from it the last node (line 15). Consequently in line 19, after the removal of this last node, we still have $path \subseteq S$ and the nodes in *path* still form a path in G.

(c) Assume that in line 8 we have: $S \in C$. Then, we need to show that in the recursive call to *ExpandSubgraph* (either in line 13 or 19), we still have $S \in C$. As for the call in line 13, observe that the last node in *path*, namely v, is in S (because we have already shown that $path \subseteq S$). This, as well as the fact that $G(S)$ is connected, implies that $G(S \cup \{u\})$ is also connected, since u is a neighbor of v. As for the call in line 19, it suffices to note that S remains unchanged compared to line 8.

(d) Let v be the last node in *path* in line 8 (in fact, this node is indeed assigned to v later on in line 9). Now, considering the following condition (we will call it Condition (d')):

(d') For every $i = 1, \ldots, |path| - 1$, the nodes that are located in $M(path[i])$ at indices: $1, \ldots, M(path[i]).getIndex(path[i+1]) - 1$ are processed.

We will prove simultaneously that Conditions (d) and (d') hold when *ExpandSubgraph* is called. In the initial calls of *ExpandSubgraph* in line 7, those conditions trivially hold, since $indexOfFirstNeighbor = 1$ and $|path| = 1$. Thus, assuming that Conditions (d) and (d') hold in line 8, we only need to prove that they hold in lines 13 and 19.

First, consider the call in line 13. Here, Condition (d) trivially holds because we have: $indexOfFirstNeighbor = 1$. It remains to show that Condition (d') also holds in line 13. At this line, *path* is the same as in line 8 except for the new node, namely u, which is now added at the end of *path*. Since Condition (d') holds in line 8, then in line 13, for every $i = 1, \ldots, |path| - 2$ the nodes that are located in $M(path[i])$ at indices $1, \ldots, M(path[i]).getIndex(path[i+1]) - 1$ are processed. It remains to show that the same holds for $i = |path| - 1$. Now since in line 13 we have: $v = path[|path| - 1]$ and $u = path[|path|]$, then we only need to show that, in line 13, the nodes that are located in $M(v)$ at indices $1, \ldots, M(v).getIndex(u) - 1$ are processed. To this end, since Condition (d) holds in line 8, then the nodes that are located in $M(v)$ at indices $1, \ldots, indexOfFirstNeighbor - 1$ are processed. As for the remaining nodes, i.e., those located

in $M(v)$ at indices: $indexOfFirstNeighbor, \ldots, M(v).getIndex(u) - 1$, each of them was considered in the previous iterations of the for-loop (lines 10-14), and was added to *Forbidden* (in line 14) unless it was already in S or in *Forbidden* (see line 12); either way the node is processed. Consequently, Condition (d') holds in line 13.

Now, consider the call in line 19. At this line, $path$ is the same as in line 8 except for the node v which is now removed from $path$. This, as well as the fact that Condition (d') holds in line 8, imply that Condition (d') also holds in line 19. It remains to show that Condition (d) also holds in line 19. At this line, w is the last node in $path$, which implies that, in line 8, w was the predecessor of v in $path$. This, as well as the fact that Condition (d') holds in line 8, imply that the nodes located in $M(w)$ at indices: $1, \ldots, M(w).getIndex(v) - 1$ are all processed in line 19. Next, we will show that the node located in $M(w)$ at index: $M(w).getIndex(v)$ is also processed in line 19. This node is v, and we will show that it is processed by showing that $v \in S$ in line 19. To this end, note that $v \in path$ in line 9, and we know from Condition (b) of Lemma 4.1 that $path \subseteq S$ in line 9. Consequently, $v \in S$ in line 9. This, as well as the fact that S in line 19 remains unchanged compared to line 9, imply that $v \in S$ in line 19, which is what we wanted to show. So far, we have shown that the nodes in $M(w)$ at indices: $1, \ldots, M(w).getIndex(v)$ are processed in line 19. This, as well as line 18, imply that in line 19 the nodes located in $M(w)$ at indices: $1, \ldots,$ $indexOfFirstNeighbor - 1$ are processed in line 19, meaning that Condition (d) holds in line 19.

(e) Let v be a node in $S \setminus path$. Then, we need to show that all the nodes in $M(v)$ are processed. To this end, note that S and $path$ initially contain the same node (see lines 6 and 7), and whenever a node is added to S it is also added to $path$ (see line 13). Thus, v must have been in $path$, and must have then been removed from $path$. This removal could only have taken place in line 15, as no other line involves removing any nodes from $path$. Based on this, it suffices to prove that when v is removed from $path$ in line 15, all the nodes in $M(v)$ are already processed. We will first prove this for the nodes located in $M(v)$ at indices: $1, \ldots,$ $indexOfFirstNeighbor - 1$, and then prove it for the nodes located in $M(v)$ at indices: $indexOfFirstNeighbor, \ldots, |M(v)|$.
   - The node v must have been the last node in $path$ before its removal from $path$ (because this removal is carried out using the operation $path.removeLast()$). This, as well as Condition (d) of Lemma 4.1, imply that in line 8 the nodes located in $M(v)$ at indices: $1, \ldots,$ $indexOfFirstNeighbor - 1$ are already processed. We need to prove that this is also the case in line 15. To this end, note that in line 15 no element was removed from S nor from *Forbidden*, compared to line 8. Therefore, if every node in $M(v)$ at indices: $1, \ldots,$ $indexOfFirstNeighbor - 1$ is already processed in line 8 (i.e., is already in either S or *Forbidden*), then this must also be the case in line 15.
   - Every node located in $M(v)$ at indices: $indexOfFirstNeighbor, \ldots, |M(v)|$ was assigned to u in line 11, which was then added to *Forbidden* in line 14, unless u was already in *Forbidden* or in S (see line 12). Thus, in line 15, every such node is already processed (i.e., is already in either S or *Forbidden*).

   With the above two points, we have shown that in line 15 all the nodes in $M(v)$ are already processed, which concludes the proof for Condition (e).

(f) Assume that in line 8 we have: $|path| \geq 1$. It remains to show that in the next recursive call to *ExpandSubgraph* (in either line 13 or 19) we also have: $|path| \geq 1$. As for line 13, $path$ is expanded compared to its initial state in line 8, and therefore: $|path| \geq 1$. As for line 19, we have $|path| \geq 1$ (this is ensured by line 16).

□

Proof of Theorem 4.2. From Condition (c) of Lemma 4.1, we know that every time function *ExpandSubgraph* is called, S induces a connected subgraph. Consequently, at the end of the current call, i.e., in line 21, S also induces a connected subgraph (because S remains unchanged compared to line 8). This concludes the proof, since the current call only outputs S in line 21, and never outputs any other subset in any other line. □

Proof of Theorem 4.3. First of all, note that the for-loop in lines 5–7 calls:

- $ExpandSubgraph(G, (v_1), \{v_1\}, \emptyset, 1)$; then
- $ExpandSubgraph(G, (v_2), \{v_2\}, \{v_1\}, 1)$; then
- $ExpandSubgraph(G, (v_3), \{v_3\}, \{v_1, v_2\}, 1)$; then
- ...
- $ExpandSubgraph(G, (v_n), \{v_n\}, \{v_1, \ldots, v_{n-1}\}, 1)$.

Thus, to prove the correctness of Theorem 4.3, it suffices to prove that the following condition holds (we will call it Condition (g)):

(g) Every time *ExpandSubgraph*(G, *path*,S,*Forbidden*,*indexOfFirstNeighbor*) is called, every $C \in \mathcal{C} : (S \subseteq C) \wedge (C \cap Forbidden = \emptyset)$ is outputted exactly once.

In so doing, we would prove that DFSE outputs, exactly once, every $C \in \mathcal{C}$ such that:

- $\{v_1\} \subseteq C$; or
- $\{v_2\} \subseteq C$ and $C \cap \{v_1\} = \emptyset$; or
- $\{v_3\} \subseteq C$ and $C \cap \{v_1, v_2\} = \emptyset$; or
- ...
- $\{v_n\} \subseteq C$ and $C \cap \{v_1, \ldots, v_{n-1}\} = \emptyset$,

which implies the correctness of Theorem 4.3. The proof proceeds by induction on $|S|$ and $|path|$. Specifically:

- In **Step 1**, we prove that Condition (g) holds when $|S| = n$, regardless of the other parameters of *ExpandSubgraph*;
- In **Step 2**, assuming that Condition (g) holds when $|S| > x$ (for some $x < n$) regardless of the other parameters of *ExpandSubgraph*, we prove that Condition (g) also holds when $|S| = x$ and $|path| = 1$ regardless of the other parameters of *ExpandSubgraph*;
- In **Step 3**, assuming that Condition (g) holds when $|S| > x$ (for some $x < n$), and when $|path| < y$ (for some $y > 1$) regardless of the other parameters of *ExpandSubgraph*, we prove that Condition (g) also holds when $|S| = x$ and $|path| = y$ regardless of the other parameters of *ExpandSubgraph*.

To visualize this inductive process, consider the illustration in Figure 7. Here:

- A circle located at (x, y) represents a call to *ExpandSubgraph* where $|S| = x$ and $|path| = y$ in line 8. Circles are only depicted at $(|S|, |path|) : 1 \leq |path| \leq |S|$; this is based on Conditions (b) and (f) of Lemma 4.1, which state that $|path| \geq 1$ and $path \subseteq S$ whenever *ExpandSubgraph* is called;

- The recursive call in line 13 is represented by a diagonal arrow pointing towards the circle at $(|S|+1, |path|+1)$; this is because the recursive call is made while adding a node to both S and $path$. Note that this recursive call can only be made if $|S| < n$ in line 8 (because otherwise there would not be any nodes to be added to S during the recursive call);
- The recursive call in line 19 is represented by an arrow pointing down towards the circle at $(|S|, |path|-1)$; the arrow is pointing downwards because, when making the recursive call, S is kept the same as in line 8, whereas the last node in $path$ has been removed compared to line 8. Note that this recursive call can only be made if $|path| > 1$ in line 8 (because otherwise lines 15 and 16 would prevent the recursive call from happening);
- Line 21 is represented by an arrow pointing down towards "**print** S"; the arrow is pointing downwards because, when executing this line, S is kept the same as in line 8, whereas the last node in $path$ has been removed compared to line 8. Note that this line can only be executed if $|path| = 1$ in line 8 (because otherwise lines 15 and 16 would prevent this line from being executed);
- The colors of the circles correspond to the steps in our inductive proof of Condition (g). In particular, **Step 1** proves that the condition holds for the calls where $|S| = n$; **Step 2** proves that it holds for the calls where $|S| < n$ and $|path| = 1$; **Step 3** proves that it holds for the calls where $|S| < n$ and $|path| > 1$.

Figure 7 can help the reader visualize the steps of the proof as they are being made. Next, we proceed with these steps.

Step 1: Assuming that in line 8 we have: $|S| = n$, we need to prove that Condition (g) holds regardless of the other parameters of *ExpandSubgraph*. First of all, since $|S| = n$, then $S = V$. This, in turn, implies that $Forbidden = \emptyset$ based of Condition (a) of Lemma 4.1. Now, let v be the last node in $path$ in line 8 (in fact, this node is indeed assigned to v later on in line 9). Since $S = V$, then all the neighbors of v are in S. Consequently, when the algorithm proceeds to the for-loop in lines 10–14, the recursive call in line 13 is never invoked, regardless of the value of *indexOfFirstNeighbor* (this is ensured by line 12). The algorithm then proceeds to line 15. Having established this fact, we now proceed by induction on $|path|$. Specifically, in **Step 1.1**, we will prove that Condition (g) holds when $|S| = n$ and $|path| = 1$ regardless of the value of *indexOfFirstNeighbor*. After that in **Step 1.2**, assuming that Condition (g) holds when $|S| = n$ and $|path| < y$ (for some $y > 1$) regardless of the value of *indexOfFirstNeighbor*, we will prove that it also holds when $|S| = n$ and $|path| = y$ regardless of the value of *indexOfFirstNeighbor*.

- **Step 1.1**: Assume that in line 8 we had: $|path| = 1$. Then, regardless of the value of *indexOfFirstNeighbor*, we will have $|path| = 0$ after the execution of line 15. This implies that the if-else statement in lines 16–21 will output S. Since the current call to *ExpandSubgraph* terminates immediately afterwards, it never outputs S again. With this, we have shown that the algorithm outputs S exactly once when $|S| = n$ and $|path| = 1$ regardless of the value of *indexOfFirstNeighbor*. Importantly, because of our assumption that $S = V$, we know that the set of connected coalitions in Condition (g), i.e., $\{C \in \mathcal{C} : (S \subseteq C) \wedge (C \cap Forbidden = \emptyset)\}$, equals $\{S\}$. Therefore, by proving that the current call to *ExpandSubgraph* outputs S exactly once when $|S| = n$ and $|path| = 1$ regardless of the value of *indexOfFirstNeighbor*, we actually prove that Condition (g) holds when $|S| = n$ and $|path| = 1$ regardless of the value of *indexOfFirstNeighbor*.
- **Step 1.2**: Assume that Condition (g) holds when $|S| = n$ and $|path| < y$ (for some $y > 1$) regardless of the value of *indexOfFirstNeighbor*. Furthermore, assume that in line 8 we had: $|path| = y$. Then, after the execution of line 15, we have: $|path| = y - 1$, which implies that

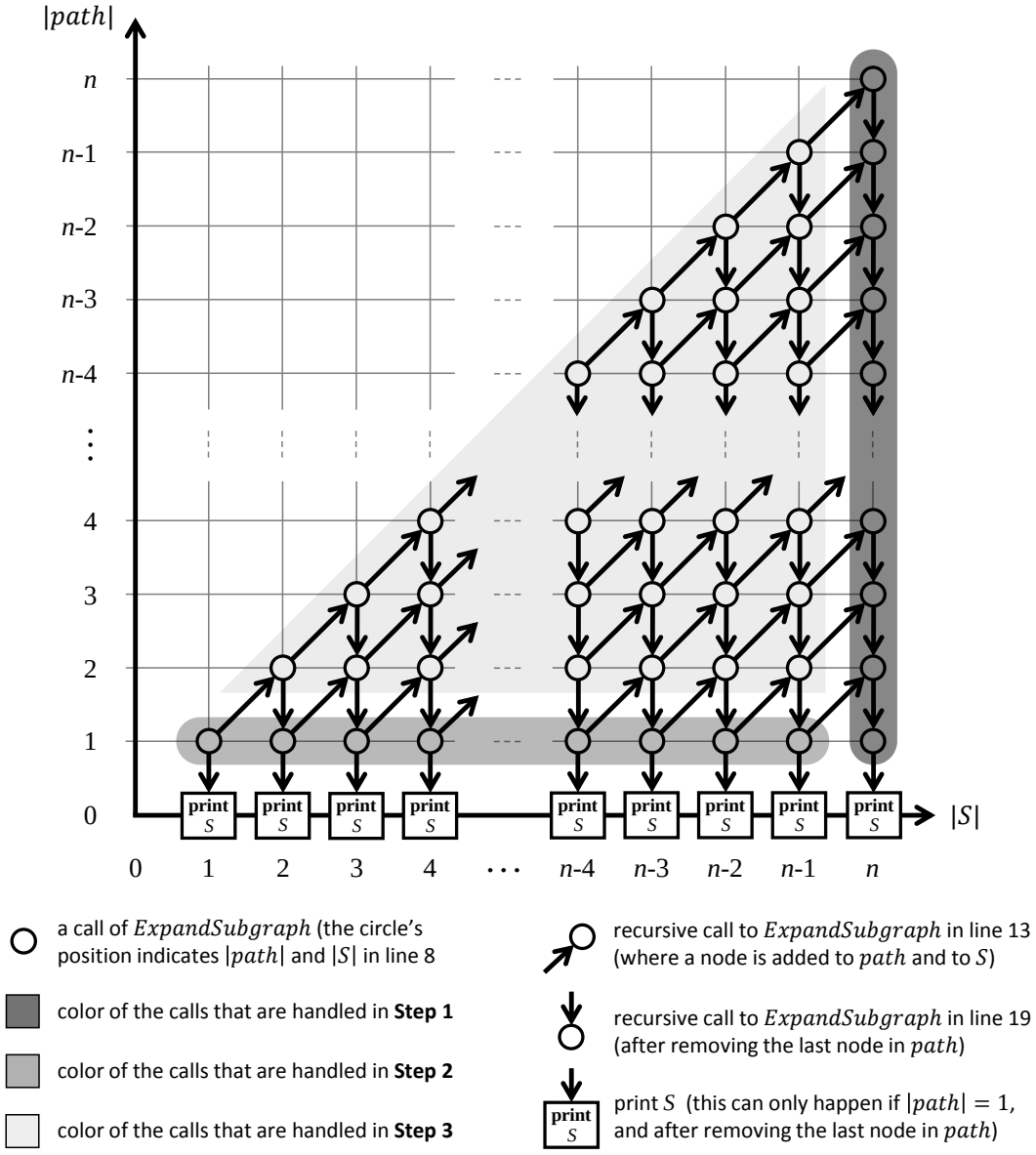


Fig. 7. In this plot, a circle located at (x, y) represents a call to *ExpandSubgraph* where $|S| = x$ and $|path| = y$ in line 8. Here, the recursive call in line 13 (which can only be made if $|S| < n$ in line 8) is represented by a diagonal arrow pointing towards the circle located at $(|S| + 1, |path| + 1)$; this is because the call is made while adding a node to both S and $path$. On the other hand, the recursive call in line 19 (which can only be made if $|path| > 1$ in line 8) is represented by an arrow pointing down towards the circle located at $(|S|, |path| - 1)$; the arrow is pointing downwards because the call is made with the same S, but after removing the last node in $|path|$. In contrast, line 21 (which can only be executed if $|path| = 1$ in line 8) is represented by an arrow pointing down towards "**print** S"; the arrow is pointing downwards because line 21 is always executed after decreasing $|path|$ to 0. The colors of the circles correspond to the steps in our inductive proof of Condition (g).

$|path| > 0$ (because $y > 1$). This, in turn, implies that the if-else statement in lines 16–21 will proceed to line 19. Here, *ExpandSubgraph* will be called with $|S| = n$ and $|path| = y - 1$, in which case Condition (g) is satisfied based on our assumption, regardless of the value of *indexOfFirstNeighbor*. We have shown that if the Condition (g) holds when $|S| = n$ and $|path| < y$ regardless of the value of *indexOfFirstNeighbor*, then it also holds when $|S| = n$ and $|path| = y$ regardless of the value of *indexOfFirstNeighbor*.

Figure 7 highlights the circles that are handled by **Step 1**; these are the ones at which $|S| = n$. Out of all these circles, the bottom one is handled by **Step 1.1**, whereas the remaining ones are handled by **Step 1.2**.

Step 2: Assuming that Condition (g) holds when $|S| > x$ (for some $x < n$) regardless of the other parameters of *ExpandSubgraph*, we need to prove that it also holds when $|S| = x$ and $|path| = 1$, regardless of the other parameters of *ExpandSubgraph*. In other words, if we write $C(S, Forbidden)$ as a shorthand for $\{C \in \mathcal{C} : (S \subseteq C) \wedge (C \cap Forbidden = \emptyset)\}$, then given a call of *ExpandSubgraph* in which $|S| = x$ and $|path| = 1$ in line 8, we need to show that this call outputs every connected coalition in $C(S, Forbidden)$ exactly once. First of all, since every coalition in $C(S, Forbidden)$ must be connected, then:

$$C(S, Forbidden) \;=\; \bigcup_{u \in N(S)} C(S \cup \{u\}, Forbidden) \;\cup\; \{S\}, \tag{14}$$

where $N(S)$ denotes the neighbors of S, i.e., $N(S) = \bigcup_{u \in S} N(u)$. Now let v be the last node in $path$ in line 8 (in fact, this node is indeed assigned to v later on in line 9). Since $|path| = 1$, then v is the only node in $path$. Based on this, as well as Conditions (a), (b), (d) and (e) of Lemma 4.1, we have:

$$\bigcup_{u \in N(S)} C(S \cup \{u\}, Forbidden) \;=\; \bigcup_{i = indexOfFirstNeighbor}^{|M(v)|} C(S \cup \{M(v)[i]\}, Forbidden). \tag{15}$$

In the for-loop (lines 10–14), for every $i \in \{indexOfFirstNeighbor, \ldots, |M(v)|\}$, the recursive call in line 13 will be made with u being equal to $M(v)[i]$ (this will happen unless $M(v)[i]$ was already in S or *Forbidden*, but in such a case the recursive call is not needed anyway, since $C(S \cup \{M(v)[i]\}, Forbidden) = \emptyset$). Now since $|S \cup \{M(v)[i]\}| > x$, then based on our assumption, if such a recursive call is made, it would output every connected coalition in $C(S \cup \{M(v)[i]\}, Forbidden)$ exactly once. Thus, based on equations (14) and (15), it remains to show that S is outputted exactly once. To this end, after the for-loop in lines 10–14, the last (and only) node in $path$ is removed in line 15. After this removal, we have: $|path| = 0$, which implies that the if-else statement in lines 16–21 will output S, after which the current call to *ExpandSubgraph* immediately terminates (this implies that S it outputted exactly once).

Step 3: Assuming that Condition (g) holds when $|S| > x$ (for some $x < n$), and when $|path| < y$ (for some $y > 1$) regardless of the other parameters of *ExpandSubgraph*, we need to prove that Condition (g) also holds when $|S| = x$ and $|path| = y$ regardless of the other parameters of *ExpandSubgraph*. In other words, if we write $C(S, Forbidden)$ as a shorthand for $\{C \in \mathcal{C} : (S \subseteq C) \wedge (C \cap Forbidden = \emptyset)\}$, then given a call of *ExpandSubgraph* in which $|S| = x$ and $|path| = y$ in line 8, we need to show that this call outputs every connected coalition in $C(S, Forbidden)$ exactly once.

First of all, note that Equation (14) holds in our case. Now, let v be the last node in $path$ in line 8 (in fact, this node is indeed assigned to v later on in line 9). Based on Conditions (a), (b), (d) and (e)

of Lemma 4.1, we have:

$$\bigcup_{u\in N(S)} C(S\cup\{u\}, Forbidden) = \bigcup_{i=indexOfFirstNeighbor}^{|M(v)|} C(S\cup\{M(v)[i]\}, Forbidden) \cup \bigcup_{u\in N(path\setminus\{v\})} C(S\cup\{u\}, Forbidden). \quad (16)$$

Having established this fact, let us now consider the for-loop in lines 10–14. More specifically, for every $i \in \{indexOfFirstNeighbor, \ldots, |M(v)|\}$, the recursive call in line 13 will be made with u being equal to $M(v)[i]$ (this will happen unless $M(v)[i]$ was already in S or *Forbidden*, but in such a case the recursive call is not needed anyway, since $C(S\cup\{M(v)[i]\}, Forbidden) = \emptyset$). Now since $|S\cup\{M(v)[i]\}| > x$, then based on our assumption, if such a recursive call is made, it would output every connected coalition in $C(S\cup\{M(v)[i]\}, Forbidden)$ exactly once. Thus, based on equations (14) and (16), it remains to show that the current call of *ExpandSubgraph* outputs, exactly once, every connect coalition in:

$$\bigcup_{u\in N(path\setminus\{v\})} C(S\cup\{u\}, Forbidden) \cup \{S\}.$$

To this end, after the for-loop in lines 10–14, the last node in *path* will be removed in line 15. After this removal, we will have: $0 < |path| < y$. Now, since $0 < |path|$, then the if-else statement in lines 16–21 will proceed to the recursive call in line 19. Here, since $|path| < y$, then based on our assumption, as well as Equation (14), we know that this recursive call will output every connected coalition in $\bigcup_{u\in N(S)} C(S\cup\{u\}, Forbidden)\cup\{S\}$ exactly once. This as well as Conditions (b), (d) and (e) of Lemma 4.1, imply that every connected coalition in $\bigcup_{u\in N(path\setminus\{v\})} C(S\cup\{u\}, Forbidden)\cup\{S\}$ will be outputted exactly once. □

Proof of Theorem 4.4. Fix $S \in C$. Let us denote by $operations(S)$ the number of operations performed in all the calls of *ExpandSubgraph* with the (third) parameter S, i.e., the calls of the form $ExpandSubgraph(G, \ldots, S, \ldots)$. We will prove that $operations(S) = O(|E|)$.

Consider the first call of the form $ExpandSubgraph(G, \ldots, S, \ldots)$ and let v be the last node in *path* in line 8. In lines 10–14, the subset of neighbors of v is considered. In particular, for each neighbor, $u \in M(v)$, if u has not been yet processed, *ExpandSubgraph* is called with the (third) parameter $S\cup\{u\}$. The operations performed in these recursive calls are calculated in $operations(S\cup\{u\})$. Thus, the number of steps performed in lines 10–14 is no larger than $|M(v)|\cdot O(1)$. Lines 15–21 require $O(1)$ steps: when line 15 is reached, node v is removed from the *path* and if *path* is not empty, then *ExpandSubgraph* is called for a shorter path and the same set S. Eventually, after such recursive call is made $|path|$ times, coalition S is printed. More precisely, if $path = (v_1, \ldots, v_k)$ in the first call of *ExpandSubgraph* for S, then by ignoring recursive calls from line 13 we get a sequence of calls:

- $ExpandSubgraph(G, (v_1, \ldots, v_k), S, Forbidden_1, 1)$,
- $ExpandSubgraph(G, (v_1, \ldots, v_{k-1}), S, Forbidden_2, M(v_{k-1}).getIndex(v_k)))$,
- ...
- $ExpandSubgraph(G, (v_1), S, Forbidden_k, M(v_1).getIndex(v_2)))$,

for some sets $Forbidden_1, \ldots, Forbidden_k \subseteq V$. Here, the last call prints S. From Theorem 4.3 and Condition (g) of its proof, we know that no more calls of the form $ExpandSubgraph(G, \ldots, S, \ldots)$

can be performed, as it would result in printing S again, and we know that S is printed only once. In result, we get:

$$operations(S) = \sum_{v \in path} |M(v)| \cdot O(1) \leq O(1) \cdot \sum_{v \in V} |M(v)| = O(|E|).$$

Note that, based on Condition (c) of Lemma 4.1, we know that $S \in C$ in every call of *ExpandSubgraph*. Thus, we calculated all steps performed in *ExpandSubgraph* calls.

It remains to calculate the number of steps performed in lines 1–7, i.e., in the body of the main function *DFSE*, without the operations performed inside the function *ExpandSubgraph*. Sorting all the nodes in line 2 takes $O(|V|log|V|)$ steps. Sorting all the neighbor lists of all nodes in lines 3–4 takes $\sum_{v \in V} O(|N(v)|^2) = O(|V| \cdot \sum_{v \in V} O(|N(v)|)) = O(|V||E|)$ steps. Finally, the for-loop in lines 5–7, without the operation inside *ExpandSubgraph* calls, requires $O(|V|)$ operations. Thus, lines 1–7 require $O(|V||E|)$ steps (since the graph is connected, we know that $O(log|V|) = O(|E|)$).

To conclude, since $|V| \leq |C|$ in every graph, the number of operations performed by DFSE is:

$$O(|V||E|) + \sum_{S \subseteq V : S \in C} operations(S) = O(|V||E|) + O(|C||E|) = O(|C||E|).$$

□

Proof of Proposition 4.5. Let us calculate the number of steps needed to enumerate all connected induced subgraphs that contain v_k but do not contain any of the nodes $v_1, \ldots, v_{k-1}$. This enumeration process starts by calling $Enumerate(G, \emptyset, \{v_k\}, \{v_1, \ldots, v_k\})$, which checks all $n-1$ edges of v_k, and puts in X all the nodes that are not in *Forbidden*, i.e., $v_{k+1}, \ldots, v_n$, and then for every non-empty subset $Y \subseteq \{v_{k+1}, \ldots, v_n\}$ calls $Enumerate(G, \{v_k\}, Y, V)$ in line 12. In every such call, all edges of all nodes in Y are considered (i.e., $|Y| \cdot (n-1)$ edges in total), but no nodes are found that are not yet processed. Thus, no further recursive calls are made to *Enumerate*. The total number of evaluated edges is then:

$$\sum_{k=1}^{n} \left((n-1) + \sum_{Y \subseteq \{v_{k+1}, \ldots, v_n\}} |Y|(n-1) \right)$$

$$= n(n-1) + (n-1) \sum_{k=1}^{n} \left((n-k)2^{n-k-1} \right) = 2^{n-1}(n^2 - 3n + 2) + (n^2 - 1).$$

□

Proof of Proposition 4.6. Since all the nodes in the graph have the same degree we assume that $V = \{v_1, \ldots, v_n\}$ and $M(v_i) = (v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$ after sorting in line 4.

First, we argue that every time *ExpandSubgraph* is called, the following conditions are satisfied:

(i) *for every $v_i, v_j \in V$, $i < j$, if v_j is processed, then v_i is processed*:
Assume v_j is processed. We know that it was added to S in line 7 or added to S or *Forbidden* in lines 13–14. In the former case, we get that $v_i \in$ *Forbidden* which proves that v_i is processed. For the later case assume v_j was added to S or *Forbidden* in lines 13–14 and let v_k be the node assigned to variable v in line 9 that precedes this call. Since $i < j$ we know that v_i is before v_j on list $M(v_k)$. Thus, from Condition (d') of the proof of Lemma 4.1, we know that, during the call, node v_i was already processed.

(ii) *if not all nodes are processed, then* $S \setminus path = \emptyset$:
Based on Condition (e) of Lemma 4.1, we know that nodes from $S \setminus path$ have all neighbors processed. Since every two nodes are neighbors in a clique, node can be in $S \setminus path$ only if all nodes are processed.

(iii) *if not all nodes processed and* $S = \{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$, $i_1 < \ldots < i_k$, *then* $path = (v_{i_1}, v_{i_2}, \ldots, v_{i_k})$:
Based on Condition (ii) we know that all nodes from S are on the path, i.e., all processed nodes which are not forbidden are on the path. Moreover, from Condition (i) we know that node with a lower id is processed before node with a higher id. That is why node with a lower id must appear before node with a higher id on the path.

In *DFSE* edges are examined in line 11. We will use the technique introduced in the proof of Theorem 4.4 and compute how many times line 11 is called when a given connected subgraph, $S \subseteq V$, is the third parameter of *ExpandSubgraph*. We denote this number by $edgeOperations(S)$. Assume $S = \{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$. From (i)–(iii) we know that in the first call where S is the third parameter $path = (v_{i_1}, v_{i_2}, \ldots, v_{i_k})$ and $Forbidden = \{v_1, \ldots, v_{i_k}\} \setminus S$. Thus, from the analysis of the proof of Theorem 4.4 and the fact that all nodes from $V \setminus \{v_1, \ldots, v_{i_k}\}$ will be added to *Forbidden* in the first call, we know that all calls where S is the third parameter form a sequence:

- $ExpandSubgraph(G, (v_{i_1}, \ldots, v_{i_k}), S, \{v_1, \ldots, v_{i_k}\} \setminus S, 1)$,
- $ExpandSubgraph(G, (v_{i_1}, \ldots, v_{i_{k-1}}), S, V \setminus S, i_k)$,
- $\ldots$
- $ExpandSubgraph(G, (v_{i_1}), S, V \setminus S, i_2)$.

In result, the number of edges examined for S is:

$$edgeOperations(S) = (n-1) + (n-i_k) + \cdots + (n-i_2)$$

Every non-empty subset of V induces a connected subgraph. Thus, summing over all $S \subseteq V$, $S \neq \emptyset$, we get:

$$\begin{aligned}\sum_{S\subseteq V, S\neq\emptyset} edgeOperations(S) &= \sum_{1\leq i_1<\ldots<i_k\leq n} (n-1) + (n-i_2) + \cdots + (n-i_k) \\ &= \sum_{S\subseteq V, S\neq\emptyset} (n\cdot|S|-1) - \sum_{1\leq i_1<\cdots<i_k\leq n} (i_1+\cdots+i_k) + \sum_{1\leq i_1<\cdots<i_k\leq n} i_1 \\ &= n^2 2^{n-1} - (2^n-1) - \sum_{j=1}^{n} j2^{n-1} + \sum_{j=1}^{n} j2^{n-j} = 2^{n-2}(n^2-n+4) - (n+1).\end{aligned}$$

□

APPENDIX: ILLUSTRATION OF THE TRÉMAUX TREE

Figure 8 presents a sample graph and its Trémaux tree.

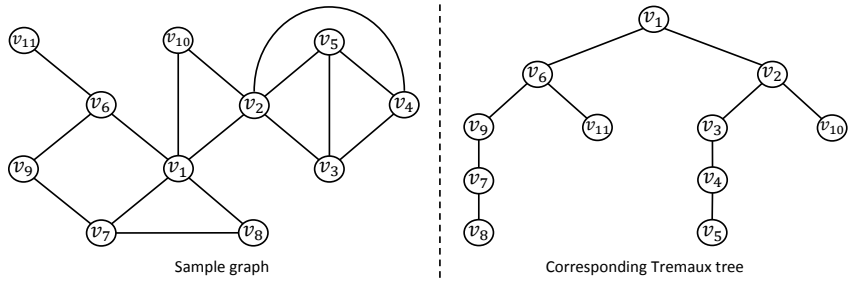


Fig. 8. A sample graph and its corresponding Trémaux tree. As is done by the DFSE algorithm, the nodes in the graph are sorted descendingly based on their degree, and are re-indexed accordingly (ties are broken uniformly at random).

APPENDIX: THE CHARACTERISTIC FUNCTIONS USED BY LINDELAUF ET AL. [26]

Lindelauf et al. [26] studied the following alternative definitions of $f_G(S)$, where ω_{ij} and ω_i denote weights of edges and nodes, respectively:

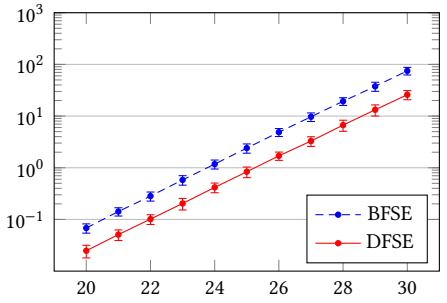
$$\begin{aligned} &\text{(i)} \quad f_G(S) = \frac{|E(S)|}{\sum_{\{v_i,v_j\}\in E(S)} \omega_{ij}}, \qquad \text{(ii)} \quad f_G(S) = \sum_{v_i\in S} \omega_i, \\ &\text{(iii)} \quad f_G(S) = \max_{\{v_i,v_j\}\in E(S)} \omega_{ij}, \quad \text{(iv)} \quad f_G(S) = \left(\max_{\{v_i,v_j\}\in E(S)} \omega_{ij}\right)\left(\sum_{v_i\in S} \omega_i\right). \end{aligned} \tag{12}$$

Different forms of function f_G reflect the fact that available data on terrorist networks differs considerably from case to case (see the appendix for more details). For instance, function (12) (i) was used to study a network of telephone communications between terrorist from the Jemaah Islamiyah Terrorist Network (reposnsible for the 2002 Bali bombing), where only the weights of edges (intensity of communication) were available but not additional intelligence on the terrorists themselves, i.e. all nodes in the network were equally weighted. For this network, Lindelauf et al. [26] proposed function (12) (i) arguing that "*A terrorist organization will try to shield its important players by keeping the frequency and duration of their interaction with others to a minimum. However, to be able to coordinate and control the attack an important player needs to maintain relationships with other individuals in the network.*" [26, p. 235]. On the other hand, function (12) (ii) was used to study the well-known 9/11 World Trade Center network of 19 nodes and 32 edges [23], where the weights of nodes indicated any additional intelligence available about individual terrorists. Here, the rationale was that the terrorists (nodes) with high weights "*play an important part in the operation. When such individuals team up, they have a significant effect on the potential success of the operation.*" [26, p. 237].

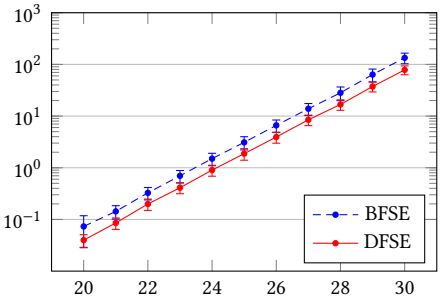
APPENDIX: EXTENDED EXPERIMENTAL ANALYSIS

Figure 9 presents the performance of the algorithms to enumerate induced connected subgraphs, i.e., DFSE and BFSE. Figure 10 presents the performance of algorithms for computing the Shapley and Myerson values, i.e., DFS-Myerson, DFS-Shapley and BFS-Shapley. As for the particular parameters, as mentioned in the main text, we base our choice on the work by Leary et al. [25] who argues that, for scale-free networks, $k = 4$ models well real-life contact networks. We also studied $k = 10$ to present the performance of our algorithm for denser networks. For small-world and Erdős-Rényi graphs we used the corresponding parameters.

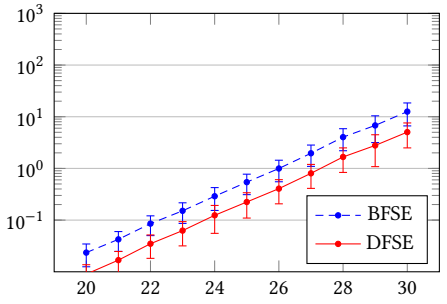
Received xxx 2018; revised xxx 2019; accepted xxx 2019



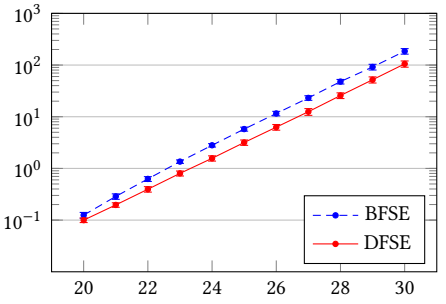
(a) Scale-free graphs (number of edges added with each node = 4).



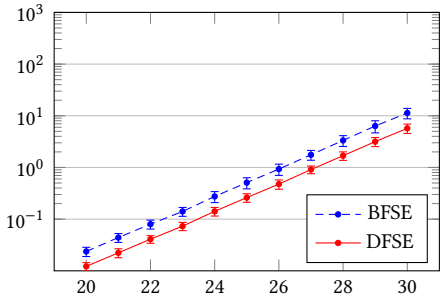
(b) Scale-free graphs (number of edges added with each node = 10).



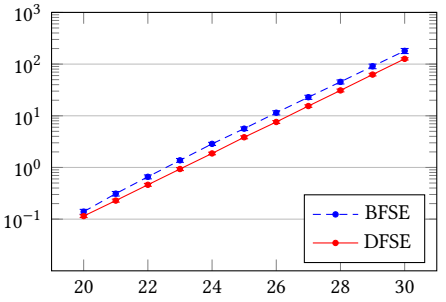
(c) Erdős-Rényi graphs (expected average degree = 4).



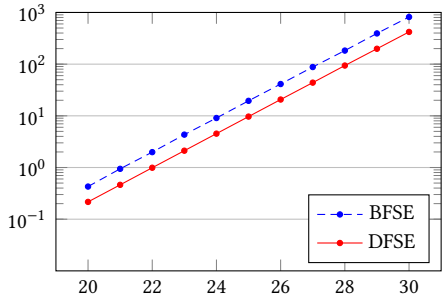
(d) Erdős-Rényi graphs (expected average degree = 10).



(e) Small-world graphs (expected degree = 4; rewiring prob. = 0.25).

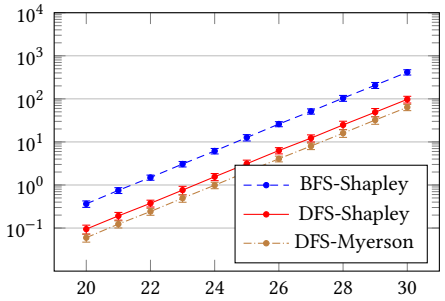


(f) Small-world graphs (expected degree = 10; rewiring prob. = 0.25).

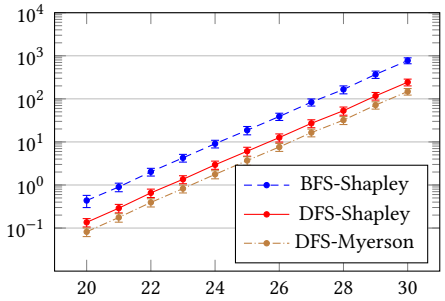


(g) Complete graphs.

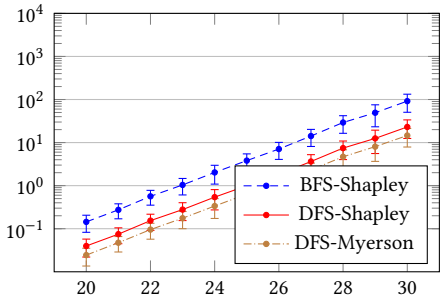
Fig. 9. Given different numbers of players (the x-axis), the figure depicts the time required to enumerate all connected induced subgraphs (the y-axis). For each random-network setting, the results are averaged over 100 different networks. DFSE was introduced in Algorithm 1. BFSE was introduced by [34] and can be found in Algorithm 2.



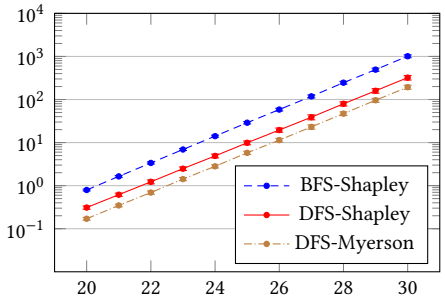
(a) Scale-free graphs (number of edges added with each node = 4).



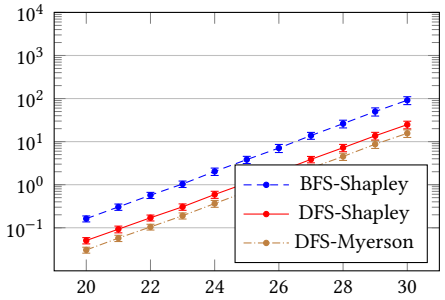
(b) Scale-free graphs (number of edges added with each node = 10).



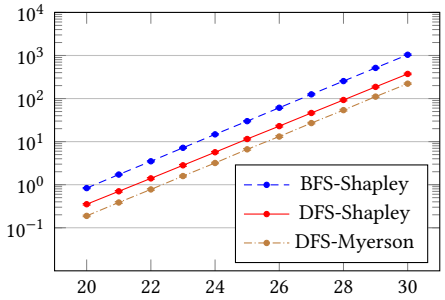
(c) Erdős-Rényi graphs (expected average degree = 4).



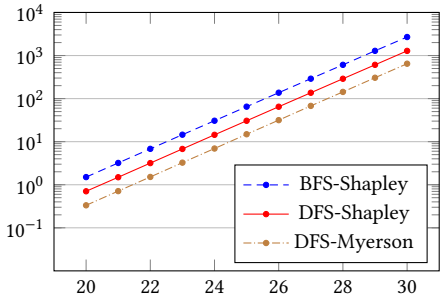
(d) Erdős-Rényi graphs (expected average degree = 10).



(e) Small-world graphs (expected degree = 4; rewiring prob. = 0.25).



(f) Small-world graphs (expected degree = 10; rewiring prob. = 0.25).



(g) Complete graphs.

Fig. 10. Given different numbers of players (the x-axis), the figure depicts the time required to compute the Shapley/Myerson value (the y-axis). For each random-network setting, the results are averaged over 100 different networks. DFS-Myerson was introduced in Algorithm 3. DFS-Shapley was introduced in Algorithm 4. BFS-Shapley comes from [33].