Enumerating Connected Subgraphs and Computing the Myerson and Shapley Values in Graph-restricted Games*

OSKAR SKIBSKI, University of Warsaw, Poland

TALAL RAHWAN, Khalifa University of Science and Technology, UAE

TOMASZ P. MICHALAK, University of Warsaw, Poland

MICHAEL WOOLDRIDGE, University of Oxford, UK

At the heart of multi-agent systems is the ability to cooperate in order to improve the performance of individual agents and/or the system as a whole. While a widespread assumption in the literature is that such cooperation is essentially unrestricted, in many realistic settings this assumption does not hold. A highly-influential approach for modelling such scenarios are graph-restricted games introduced by Myerson [36]. In this approach, agents are represented by nodes in a graph, edges represent communication channels, and a group can generate an arbitrary value only if there exists a direct or indirect communication channel between every pair of agents within the group. Two fundamental solution-concepts that were proposed for such games are the *Myerson value* and the *Shapley value*. While an algorithm has been developed to compute the Shapley value in arbitrary graph-restricted games, no such general-purpose algorithm has been developed for the Myerson value to date. With this in mind, we set to develop for such games a general-purpose algorithm to compute the Myerson value, and a more efficient algorithm to compute the Shapley value. Since the computation of either value involves enumerating all connected induced subgraphs of the game's underlying graph, we start by developing an algorithm dedicated to this enumeration, and show empirically that it is faster than the state of the art in the literature. Finally, we present a sample application of both algorithms, in which we test the Myerson value and the Shapley value as advanced measures of node centrality in networks.

CCS Concepts: • **Theory of computation → Solution concepts in game theory**; **Network games**;

Additional Key Words and Phrases: Coalitional Games, Myerson value, Depth-First Search, Algorithms

ACM Reference Format:

Oskar Skibski, Talal Rahwan, Tomasz P. Michalak, and Michael Wooldridge. 2018. Enumerating Connected Subgraphs and Computing the Myerson and Shapley Values in Graph-restricted Games. *ACM Trans. Intell. Syst. Technol.* 1, 1, Article 1 (January 2018), 42 pages. https://doi.org/10.1145/3235026

*This article is an extended version of a paper originally presented at AAMAS-14 [47] with a number of new technical results. See Acknowledgements for details.

Authors' addresses: Oskar Skibski, University of Warsaw, Banacha 2, Warsaw, 02-097, Poland, oskar.skibski@mimuw.edu.pl; Talal Rahwan, Khalifa University of Science and Technology, Abu Dabi, UAE, talal.rahwan@ku.ac.ae; Tomasz P. Michalak, University of Warsaw, Banacha 2, Warsaw, 02-097, Poland, tomasz.michalak@mimuw.edu.pl; Michael Wooldridge, University of Oxford, OX1, 3QD, Oxford, UK, michael.wooldridge@cs.ox.ac.uk.

2157-6904/2018/1-ART1 $15.00

https://doi.org/10.1145/3235026

1 INTRODUCTION

At the heart of multi-agent systems is the ability to coordinate activities in order to improve the performance of individual agents and/or the system as a whole [55]. Among the many coordination paradigms that were studied in the literature [19], one that received considerable attention is *Coalition Formation*, whereby the agents (or "*players*") divide themselves into groups (or "*coalitions*") so that members of the same coalition coordinate their activities, and possibly agree on how to divide the payoff from cooperation among themselves. One of the fundamental payoff-division schemes extensively studied to date is the *Shapley value* [44] due to its various desirable properties.

A widespread assumption in the literature on coalition formation is that cooperation is essentially unrestricted [9], i.e., any coalition can be created and may have an arbitrary payoff (or "*value*"). However, there are many realistic settings where this assumption does not hold. Often, the players can communicate and coordinate their activities only via certain communication channels [52]. Such restrictions emerge in a variety of domains, including, but not limited to, social networks [20], political alliances [37], and sensor and communication networks [41].

A highly-influential approach for modelling such scenarios was introduced by Myerson [36], who described a coalitional game over a graph, G, in which nodes represent players and edges represent available communication channels. In this setting, a coalition can have an arbitrary value only if it induces a connected subgraph of G; in this case the coalition itself is said to be *connected*. Intuitively, members of the same coalition are able to coordinate their activities only if there exists either a *direct* communication channel (i.e., an edge) or an *indirect* communication channel (i.e., a path) between them in the coalition. To appreciate the rationale, consider two players who are in coalition C but are not able to communicate with one another (neither directly through an edge between them, nor indirectly through other members of C). Since such players have no means of coordinating their activities, it is reasonable to assume that there is no value added from putting them in the same coalition. This model is now widely known as a *graph-restricted game* [36]. As restrictions in communication occur naturally in multi-agent systems, the graph-restricted games and the related solution concepts have recently attracted increasing attention in the literature [12, 49]. The applications related to multi-agent technologies include the aforementioned sensor and communication networks [41], logistics [21], smart energy and ridesharing [7].

One can make different assumptions on how the value of a disconnected coalition is computed in a graph-restricted game. In particular:

- **Myerson game**: in his seminal work, Myerson [36] assumed that the value of a disconnected coalition, C, equals the sum of values of the connected coalitions whose union yields C. Arguably, although disconnected coalitions cannot communicate nor coordinate their activities, there can still work as separate coalitions. In this class of games—now known as *Myerson games*—Myerson [36] famously showed that the Shapley value is uniquely determined by certain graph-related properties (see Section 3). In recognition of his result, the Shapley value in a Myerson game is now widely known as the *Myerson value*.
- **Connectivity games**: Amer and Giménez [2] assumed that every disconnected coalition has a value of 0. This class of games is known as *connectivity games*. The authors focused on a subclass where the value of a connected coalition equals 1; we will refer to this subclass as 0-1-*connectivity game*. Later on, Lindelauf et al. [26] considered the more general case where a connected coalition can have an arbitrary value. Both of the aforementioned studies have used the Shapley value as a payoff division scheme.

To compute either the Myerson value in a Myerson game, or the Shapley value in a connectivity game, the original definitions of those values require visiting every possible coalition, be it connected or otherwise. Bilbao [6] proposed an improved method for computing the Myerson value in a Myerson game, and Michalak et al. [33] proposed an improved method for computing the Shapley value in a connectivity game. In both methods, the improvement comes from avoiding the need to visit the disconnected coalitions. Technically, those methods require the following fundamental graph operations:

- the *enumeration of all connected induced subgraphs* of the game's underlying graph;
- the *identification of cut vertices* in each enumerated subgraph, where a cut vertex is a node whose removal disconnects the subgraph (see Section 3 for a formal definition).

As such, the efficiency of computing the Myerson value (in a Myerson game) and the Shapley value (in a connectivity game) depends on how well those two operations are combined together. Due their importance, both operations have been extensively studied in graph theory, and there already exist highly optimized algorithms to carry out each of them separately. In particular, the state-of-the-art algorithm for enumerating all connected induced subgraphs is due to Moerkotte and Neumann [34]; this algorithm is based on *breadth-first search*. In contrast, the state-of-the-art algorithm for identifying cut vertices is due to Hopcroft and Tarjan [18]; this algorithms is based on *depth-first search*. Unfortunately, given that the two algorithms are based on orthogonal design principles, combining them efficiently is challenging [33].

With this in mind, the contributions of this article can be summarised as follows:

- In Section 4, we propose a new algorithm for enumerating all connected induced subgraphs in an arbitrary graph. Unlike the alternative algorithm by Moerkotte and Neumann [34] which uses breadth-first search, our algorithm uses depth-first search, making it compatible with depth-first-search algorithm for identifying cut vertices by Hopcroft and Tarjan [18]. More important, our enumeration algorithm turns out to be faster in practice than that of Moerkotte and Neumann, making ours the fastest algorithm for enumerating connected induced subgraphs to date (the empirical evaluation can be found in Section 7).
- In Section 5, we propose a new formula for the Myerson value which, just like the formula by Bilbao [6], considers only the connected subgraphs but, unlike this formula, it does not require the identification of cut vertices. Building upon this formula and our enumeration algorithm, we propose the first dedicated algorithm to compute the Myerson value in arbitrary graphs.
- Building again upon our enumeration algorithm, in Section 6 we propose a new algorithm to compute the Shapley value in connectivity games. Since our algorithm traverses the graph in depth-first order, it is able to efficiently identify cut vertices. We empirically show in Section 7 that our algorithm outperforms that of Michalak et al. [33]; hence, our algorithm is the state of the art for computing the Shapley value in connectivity games.
- Finally, in Section 8, we present a sample application of the algorithms, where we compare the performance of the Shapley value and the Myerson value as the tools to identify key figures in terrorist networks. While, due to computational limitations, the existing literature studied only a few instantiations of this approach based on the Shapley value, with our algorithms we can perform a much wider analysis.

Note that all three algorithms proposed in this article perform in the worst case the exponential number of steps. This is because the number of all connected induced subgraphs can be exponential in the number of nodes, which implies that both the output for enumeration algorithm and the input for computing the Myerson and Shapley values are also exponential.

2 RELATED WORK

In this section, we first discuss the body of works on enumerating the connected induced subgraphs and identifying cut vertices. After that, we discuss the literature on the computational aspects of the Shapley value and the Myerson value in graph restricted games.

The enumeration of connected induced subgraphs is one of the fundamental algorithmic operations in graph theory, with applications as diverse as computing topological indices for molecular graphs [38], optimizing cost-based queries [34], and searching for optimal coalition structures in graph restricted games [53]. A few dedicated algorithms have been proposed for this purpose in the literature. The early works include the reverse-search algorithm by Avis and Fukuda [4] and Brüngger et al. [8], and the breadth-first search algorithm by Sharafat and Marouzi [45]. These algorithms, however, performed numerous redundant operations. This shortcoming was rectified by the highly-optimized breadth-first search algorithm of [34]. To date, this algorithm remains the state-of-the-art for enumerating the connected induced subgraphs.[1]

As we will show, one of the important features of the algorithm proposed in this article is that it not only offers a slightly better performance but it has a depth-first search structure which allows for an efficient combination with other algorithms of this type. In result, it allows for solving problems that cannot be efficiently solved using the breadth-first search algorithm. This includes the problem of identifying cut vertices, especially important in the context of computing the Shapley value for connectivity game, for which the state-of-the-art algorithm based on the depth-first search was proposed by Hopcroft and Tarjan [18]. See Section 6 for more details.

Bilbao [6] was the first to propose the explicit formulas for the Myerson value that traverse only connected induced subgraphs. Later on, a different closed-from expression was proposed by Elkind [12]. However, both these methods, on top of traversing connected induced subgraphs, also require finding all the cut vertices in every such subgraph. Algaba et al. [1] proposed a method of computing the Myerson value based on Harsanyi's dividends in time $O(3^n)$ for n players. They also developed polynomial-time algorithms for special classes of graphs. Furthermore, the computational properties of the Myerson value for some special classes of games were considered in [14, 17].

Another related body of literature concerns centrality analysis, where the aim is to rank nodes in a network in the way that fits best a particular application. The most well-known centrality indices are degree, closeness, and betweenness centralities. In particular, degree centrality quantifies the importance of a node by the number of its incident edges. Closeness centrality highlights nodes that are close to all other nodes in the network. Betweenness centrality counts the shortest paths between any two nodes in the network, and ranks nodes according to the number of the shortest paths they belong to. Apart from many refinements and extensions of these three standard centrality measures, various other concepts have been proposed in the literature. Some of the advanced centrality measures are based on game-theoretic solution concepts. In particular, a number of more-advanced centrality measures are either directly built upon Myerson's graph-restricted game or inspired by it. The first work in this line of research was due to Gómez et al. [16] who defined the centrality of a node v_i to be the difference between the Shapley value of v_i (which is independent of the network topology) and the Myerson value of v_i (which takes the network topology into consideration). Thus, by interpreting the Shapley value as a power index in a coalitional game, Gomez et al.'s Shapley/Myerson-based centrality measure represents the increase (or decrease) in v_i's power due

[1] The same breadth-first search algorithm was rediscovered by Voice et al. [53]. We also refer the reader to more recent papers where the authors proposed algorithms to enumerate induced connected subgraphs of specific types: of a given size [22] or with the value assigned by an external function, $f : C \to \mathbb{R}$, higher than the specified threshold [28]. Adapting these algorithms to our problem would not be efficient.

to its position in the network. In turn, del Pozo et al. [11] proposed a measure for directed graphs, based on generalised coalitional games [39], where the value of a coalition is influenced by the sequence in which the players have joined it. More specifically, the authors adapted these games to networks in the spirit of Myerson and then based their measure on a parametric family of solution concepts that include two extensions of the Shapley value to generalised coalitional games: one by Nowak and Radzik [39] and the other by Sánchez and Bergantiños [43]. In a similar spirit, Amer et al. [3] defined a game-theoretic centrality measure called accessibility in oriented networks.

Skibski et al. [48] proposed *the attachment centrality*—the first game-theoretic centrality measure obtained using the axiomatic approach. This measure is the Myerson value of the following game: $f(S) = 2(|S| - 1)$. Later on, the attachment centrality was extended to weighted graphs [50]. Skibski et al. [46] performed the axiomatic analysis of game-theoretic centralities. Michalak et al. [31] studied the computational properties of game-theoretic centralities based on the Shapley value, but not on the Myerson value.

Finally, we point the reader to works from the computer science literature, where graph-restricted games were considered [29, 53].

3 PRELIMINARIES

This section starts by presenting some necessary concepts from *graph theory* and *cooperative game theory*, before formally introducing the *Shapley value* and the *Myerson value*. A table describing the main notation can be found in the appendix.

3.1 Graph-Theoretic Concepts

A graph, $G = (V, E)$, consists of a set of vertices (or nodes), $V = \{v_1, \ldots, v_n\}$, and a set of edges, $E \subseteq V \times V$. Arbitrary nodes will often be denoted by either v, u or w, although we may sometimes use the notation v_i or v_j instead. For any node, $v \in V$, we denote by $N(v)$ the set of neighbors of v. A path is a sequence of distinct nodes in which every two consecutive nodes are connected by an edge. A graph is said to be *connected* if and only if there exists a path between every pair of nodes in that graph. We say that $v \in V$ is a *cut vertex* in a connected graph, (V, E), if the removal of v (along with all its edges) splits the graph into disconnected components, i.e., if the graph $(V \setminus \{v\}, E \setminus \{\{v, w\} \in E : w \in V\})$ is disconnected.

For any given subset of nodes, $S \subseteq V$, we will denote by $E(S) \subseteq E$ the set of all edges between members of S. Formally, $E(S) = \{\{v, w\} \in E : v, w \in S\}$. The subgraph of G *induced* by S is $(S, E(S))$; we will denote this subgraph by $G(S)$. Since the subgraphs considered in this article are all induced (but not necessarily connected), we will often omit the term "*induced*" when there is no risk of confusion. Note that if $G(S)$ is disconnected, then surely it consists of two or more connected components; we will denote by $\mathcal{K}(S) = \{K_1, K_2, \ldots, K_m\}$ the subsets of S that induce those components. This means that $\mathcal{K}(S)$ is a partition of S, and that for every $K_i \in \mathcal{K}(S)$, the subgraph $G(K_i)$ is connected.

3.2 Graph-Restricted Games

The *players* in our setting represent (or are represented by) the nodes in a graph, $G = (V, E)$. As such, we interpret V as the set of players. A coalition, i.e., a subset of player, $S \subseteq V$, is said to be connected if and only if $G(S)$—the subgraph of G induced by S—is connected; otherwise the coalition is said to be disconnected. The set of all connected coalitions will be denoted by $\mathcal{C}$. We

will restrict our attention to games in *characteristic function form*. Such games are each defined by a set of players—V in our case—and a *characteristic function*, $f : 2^V \to \mathbb{R}$, that assigns to every coalition (connected or otherwise) a real number representing its *payoff*, or "*value*".

A *graph-restricted game* is defined over an underlying graph, $G = (V, E)$, whose nodes represent the players of the game. The general idea behind these games is that a coalition can achieve arbitrary synergy (i.e., it can have an arbitrary value) only if that coalition happens to be connected in G. As such, whether a coalition is connected or not has a direct bearing on that coalition's value. We will focus on two subclasses of graph-restricted games. In particular:

- Myerson [36] assumed that a connected coalition may have an arbitrary value, which is specified by a function, $f_G : C \to \mathbb{R}$. On the other hand, the value of a disconnected coalition, $S \in 2^V \setminus C$, is the sum of the values of the coalitions in $\mathcal{K}(S)$, i.e., those that induce the connected components in $G(S)$.[2] This leads to the following characteristic function:
$$f_G^{\mathcal{M}}(S) = \sum_{K_i \in \mathcal{K}(S)} f_G(K_i), \tag{1}$$
where $\mathcal{M}$ stands for Myerson. Such a game will be referred to as a *Myerson game*.
- Amer and Giménez [2] formalised an alternative model in which the value of a connected coalition is 1 whereas the value of a disconnected coalition is 0. This produces a *simple game*:[3] whose characteristic function takes the following form:
$$f_G^{\mathcal{A}}(S) = \begin{cases} 1 & \text{if } S \in C \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$
where $\mathcal{A}$ stands for Amer and Giménez. We refer to such a game as a 0-1-*connectivity game*. Lindelauf et al. [26] generalized this game, by allowing the connected coalitions to have arbitrary values. This produced the following characteristic function:
$$f_G^{\mathcal{L}}(S) = \begin{cases} f_G(S) & \text{if } S \in C \\ 0 & \text{otherwise,} \end{cases} \tag{3}$$
where $\mathcal{L}$ stands for Lindelauf et al., and $f_G : C \to \mathbb{R}$. A game with such a characteristic function will be referred to as a *connectivity game*, which is of course a generalization of a 0-1-connectivity game.

3.3 The Shapley Value

Assuming that the grand coalition (i.e., the largest possible coalition, V) is formed, a "*solution*" to a coalitional game specifies a way to divide $f(V)$ among the players so as to meet certain criteria. One desirable criterion that is often sought after is *fairness*. Typically, in the context of coalitional games, fairness is assessed based on the degree to which each player's payoff reflects its contribution to the coalition. To date, the best-established solution concept that captures this notion of fairness is the *Shapley value* [44][4]. This solution concept has been shown to satisfy various desirable properties; for more details, see the work by Maschler et al. [27]. At its heart, the Shapley value is based on the notion of *marginal contribution*. Specifically, given a game in characteristic function form, (V, f),

[2]Note that if $S \in C$ then $\mathcal{K}(S) = \{S\}$.
[3]Simple coalitional games constitute a popular class of games in which every coalition has a value of either 1 or 0 [9].
[4]More precisely, a solution concept is a rule of assigning payoff according to the payoff division scheme, e.g., the Shapley value, and not the scheme itself. Nevertheless, as usual in the literature, we will refer to the Shapley value and the Myerson value as solution concepts.

the marginal contribution of a player, $v_i \in V$, to a coalition $S \subseteq V \setminus \{v_i\}$ is the difference in value that v_i creates when joining S, i.e., it is: $f(S \cup \{v_i\}) - f(S)$. The Shapley value then divides $f(V)$ by assigning to each player, $v_i \in V$, a weighted average of all its marginal contributions, taking into consideration every coalition that this player could possibly belong to; this payoff is called *the Shapley value of* v_i, and is denoted by $SV_i(f)$. More precisely, $SV_i(f)$ is computed as follows:

$$SV_i(f) = \sum_{S \subseteq V, v_i \in S} \frac{(|S|-1)!(|V|-|S|)!}{|V|!} \big(f(S) - f(S \setminus \{v_i\})\big). \tag{4}$$

Equivalently, the above formula of the Shapley value of player v_i can be rewritten as follows:

$$SV_i(f) = \frac{1}{|V|!} \sum_{\pi \in \Pi} \big(f(S_i^\pi) - f(S_i^\pi \setminus \{v_i\})\big), \tag{5}$$

where Π is the set of all permutations of V, whereas S_i^π is the coalition consisting of v_i and all the players that precede v_i in the permutation π. Viewed from this perspective, the Shapley value of a player, $v_i \in V$, equals its average marginal contribution to the players that precede it in an arbitrary permutation.

3.4 The Myerson Value

A celebrated result of Myerson [36] is that, when applying the Shapley value to a Myerson game, $(V, f_G^{\mathcal{M}})$, we obtain a payoff division scheme that uniquely satisfies certain desirable properties. Nowadays, whenever the Shapley value is applied to a Myerson game, it is widely referred to as the *Myerson value*, and is often denoted differently than the Shapley value; we will denote it by $MV_i(f_G)$ for player v_i, i.e., $MV_i(f_G) = SV_i(f_G^{\mathcal{M}})$, where f_G is the function that assigns values to connected coalitions.

In the next three sections we will present our algorithms, starting with the enumeration of the connected induced subgraphs, followed by the computation of the Myerson value in Myerson games, and lastly the computation of the Shapley value in connectivity games.

4 DFS ENUMERATION OF CONNECTED INDUCED SUBGRAPHS

We start in Section 4.1 by presenting our *Depth-First-Search Enumeration (DFSE)* algorithm for enumerating all connected induced subgraphs. Then in Section 4.2, we discuss some of the properties of our algorithm. Finally, in Section 4.3, we compare the workings of our enumeration algorithm against those of its counterpart by Moerkotte and Neumann [34], which we refer to as *Breadth-First-Search Enumeration (BFSE)*. As mentioned earlier, our enumeration algorithm will be used in subsequent sections as the cornerstone upon which we build our algorithms for computing the Myerson value in arbitrary Myerson games and the Shapley value in connectivity games.

4.1 The DFSE Algorithm

In this subsection we outline our Depth-First-Search Enumeration (DFSE) algorithm, which enumerates all connected induced subgraphs of an arbitrary graph. In what follows, every connected induced subgraph is represented by its set of nodes. As such, we may write, e.g., "subgraph $\{v_i, v_j\}$" instead of "subgraph $G(\{v_i, v_j\})$".

In a nutshell, our algorithm starts with a single-node subgraph, namely $S = \{v_i\}$, and tries to expand it—by adding one node at a time—while ensuring that the subgraph remains connected. In particular, for every added node, the algorithm examines each of its neighbors, and whenever a

ALGORITHM 1: The Depth-First-Search Enumeration (DFSE) algorithm for enumerating all connected induced subgraphs

Input: Graph $G = (V, E)$
Output: Every connected induced subgraph of G

```
1  DFSE begin
2      Sort V by degree descendingly, and re-index the nodes accordingly;
3      for i ← 1 to |V| do
4          Sort N(v_i) by degree descendingly, and assign the sorted list to parameter M(v_i);
5      for i ← 1 to |V| do
6          S ← {v_i};
7          ExpandSubgraph(G, (v_i), S, {v_1, ..., v_{i-1}}, 1);
8  ExpandSubgraph(G, path, S, Forbidden, indexOfFirstNeighbor) begin
9      v ← path.last();
10     for indexOfCurrentNeighbor ← indexOfFirstNeighbor to |M(v)| do
11         u ← M(v).get(indexOfCurrentNeighbor);
12         if (u ∉ S) ∧ (u ∉ Forbidden) then
13             ExpandSubgraph(G, path ∪ {u}, S ∪ {u}, Forbidden, 1);
14             Forbidden ← Forbidden ∪ {u};
15     path.removeLast();
16     if path.length() > 0 then
17         w ← path.last();
18         indexOfFirstNeighbor ← M(w).getIndex(v) + 1;
19         ExpandSubgraph(G, path, S, Forbidden, indexOfFirstNeighbor);
20     else
21         print S;
```

neighbor, u, is found that has not yet been visited by the algorithm, the process is divided into two processes: the first expands the subgraph with u, whereas the second process marks u as *forbidden* (which ensures that u is never considered later on by this process) and looks for another neighbor that has not yet been visited by the algorithm, and so on.

The pseudocode of DFSE is presented in Algorithm 1. The main function—*DFSE* (lines 1–7)—starts by sorting all nodes descendingly based on their degrees, and then re-indexes them accordingly, i.e., v_1 becomes the node with the highest degree, v_2 the one with the second highest degree, and so on. Next, the neighbors of every node, $v_i \in V$, are sorted based on their degree descendingly (lines 3–4); the sorted list is assigned to a parameter, $M(v_i)$. The two main operations to handle this parameter are: $M(v_i).get(j)$ which returns the neighbor at index j in $M(v_i)$; and $M(v_i).getIndex(v_j)$ which returns the index of neighbor v_j in $M(v_i)$.[5] In lines 5–7, for every node, $v_i \in V$, the algorithm creates the single-node subgraph $\{v_i\}$ and starts the process of expanding this subgraph by calling the recursive function *ExpandSubgraph*. The remainder of the algorithm description focuses on this particular function.

In the recursive function *ExpandSubgraph*, the parameter *path* is a sequence representing the current search path during the depth-first traversal of the graph. The three main operations to

[5] As the function $getIndex$ is called multiple times, the information that it returns can be pre-computed in the main function, $DFSE$, and stored in a table to enable constant-time access to this information.

handle this parameter are: $path.length()$ which returns the number of nodes in the path, $path.last()$ which returns the last node in the path, and $path.removeLast()$ which removes that node from the path. To improve readability, we will sometimes use set notation when dealing with a sequence such as $path$, e.g., we may write $path \cup \{u\}$ to denote the sequence that results from adding u to the end of $path$, or write $path \subseteq S$ to express the fact that all the nodes in $path$ belong to S, or write $u \in S \setminus path$ to express the fact that u is in S but not in $path$. The first line in the function *ExpandSubgraph* involves retrieving the last node in $path$, and assigning it to v. Then in lines 10–14, for every neighbor of v that has not been visited earlier, the function checks whether this neighbor is already in S and checks whether it is marked as a forbidden node (see line 12). If it is neither, then the neighbor is added to S—through a recursive call to *ExpandSubgraph* in line 13—before being marked as forbidden. Having processed all the neighbors of v, in lines 15–17 the function backtracks to v's predecessor, called w. Next, in lines 18 and 19, the function deals with the neighbors of w that have not yet been processed. Finally, when $path$ is empty, the algorithm prints the subgraph S (line 21). An illustration of DFSE can be found in the appendix.

Finally, we note that the runtime of the algorithm is affected in practice by the sequence in which the nodes are processed. Out of all the sequences that we have experimented with, sorting the nodes (and their neighbors) descendingly based on their degrees seemed to yield the best performance. Based on this, in lines 2–4, the nodes are sorted accordingly. Nevertheless, identifying an optimal such sequence remains an open question.

Having described the DFSE algorithm, in the following subsection we analyse its properties and prove its correctness.

4.2 Properties of the DFSE Algorithm

In this subsection, we discuss the key properties of the DFSE algorithm, namely:

- *soundness*—for every graph, DFSE enumerates *only* connected induced subgraphs;
- *completeness*—for every graph, DFSE enumerates *all* connected induced subgraphs;
- *non-redundancy*—for every graph, DFSE never enumerates the same connected induced subgraph *more than once.*

We start by stating basic properties of the parameters in the recursive function *ExpandSubgraph* in Algorithm 1.

Lemma 4.1. *Every time ExpandSubgraph*$(G, path, S, Forbidden, indexOfFirstNeighbor)$ *is called, the parameters therein satisfy the following conditions:*

(a) *The sets S and Forbidden are disjoint;*

(b) *All nodes in the parameter path belong to S and form a path in G;*

(c) *S induces a connected subgraph, i.e.,* $S \in C$*;*

(d) *Let v be the last node in the parameter path. Then, the nodes located in* $M(v)$ *at indices:* $1, \ldots,$ $indexOfFirstNeighbor - 1$ *are "processed", i.e., each of them is either in S or in Forbidden;*

(e) *For every* $v \in S \setminus path$*, all the neighbors of v are "'processed", i.e., each of them is either in S or in Forbidden;*

(f) $|path| \geq 1$.

Based on Lemma 4.1, we can prove the soundness, completeness and non-redundancy of the DFSE algorithm.

ALGORITHM 2: The Breadth-First-Search Enumeration (BFSE) algorithm for enumerating all connected induced subgraphs

Input: Graph $G = (V, E)$
Output: Every connected induced subgraphs of G

```
1  BFSE begin
2      for i ← 1 to |V| do
3          Enumerate(G, ∅, {v_i}, {v_1, ..., v_i});
4  Enumerate(G, Old, New, Forbidden) begin
5      print Old ∪ New;
6      X ← ∅; foreach v ∈ New do
7          foreach u ∈ N(v) do
8              if u ∉ Forbidden ∪ X then
9                  X ← X ∪ {u};
10     foreach Y ⊆ X : Y ≠ ∅ do
11         Enumerate(G, Old ∪ New, Y, Forbidden ∪ X);
```

Theorem 4.2. *The DFSE algorithm is sound, i.e., for every graph, it enumerates* only *connected induced subgraphs.*

Theorem 4.3. *For every graph, the DFSE algorithm enumerates every connected induced subgraph exactly once.*

Finally, we observe that the time complexity of DFSE is linear in the number of connected subgraphs.

Theorem 4.4. *The time complexity of DFSE is $O(|C||E|)$ for a connected graph.*

The proofs are available in the appendix.

4.3 DFS versus BFS Enumeration of Connected Induced Subgraphs

To date, the state-of-the-art algorithm for enumerating all connected induced subgraphs is due to Moerkotte and Neumann [34]. As opposed to our algorithm, which traverses the graph in a depth-first manner, their algorithm uses breadth-first search. As such, we will refer to their algorithm as BFSE (Breadth-First-Search Enumeration), the pseudocode of which is presented in Algorithm 2. Specifically, in the i^{th} step of the main function, *BFSE*, the algorithm enumerates every subgraph that contains v_i and does not contain the any of the nodes $v_1, \ldots, v_{i-1}$. To this end, the recursive function, *Enumerate*, is called with four parameters: (i) the graph G; (ii) an *Old* part of the subgraph; (iii) a *New* part of the subgraph; and (iv) the set of all nodes that we already considered, denoted by *Forbidden* (roughly speaking, this set consists of the nodes that are already in subgraph, as well as the nodes that we have considered but did not include in the subgraph). Now, *Enumerate* outputs the current subgraph, $Old \cup New$, and tries to enlarge it. In order to do that, it lists all not-yet considered neighbors (the set of which is denoted by X) and for every subset $Y \subseteq X$ it calls *Enumerate* with the subgraph enlarged by Y and the set of considered nodes expanded by X.[6] An illustration of BFSE can be found in the appendix.

[6]Our pseudocode of BFSE is more efficient than the original. Specifically, if we follow the original pseudocode and merge both parts of the subgraph (Old and New) in the declaration (and calls) of $Enumerate$, then to find the neighbors in lines 7–12 we would have to consider also nodes from the old part of the subgraph. This is clearly redundant, as all their neighbors are already in the set *Forbidden*.

Next, we provide two propositions showing that for n-cliques BFSE examines approximately twice the number of edges compared to our DFSE algorithm (later on in Section 7 we compare the two algorithms empirically, given a wide range of graphs).

Proposition 4.5. *Given an n-clique, BFSE examines* $2^n(n^2/2 + O(n))$ *edges.*

Proposition 4.6. *Given an n-clique, DFSE examines* $2^n(n^2/4 + O(n))$ *edges.*

5 COMPUTING THE MYERSON VALUE IN ARBITRARY MYERSON GAMES

In this section we present the first dedicated algorithm for computing the Myerson value in an arbitrary Myerson game. The complexity of this computation may vary depending on the definition of f_G—the function that specifies the value of every connected coalition. We will treat f_G as an oracle and assume that it returns the value of any connected coalition in constant time.

In the following theorem we prove that, in order to calculate the Myerson value of player in a Myerson game, *it suffices to examine only the connected coalitions*. Furthermore, unlike the case with the Shapley value, calculating the Myerson value does not require the identification of cut vertices when enumerating the connected coalitions.

Theorem 5.1. *For any graph, G, and any* $v_i \in V$*, and any* $f_G : C \rightarrow \mathbb{R}$*, the following holds:*

$$MV_i(f_G) = \sum_{S \in C, v_i \in S} \frac{(|S|-1)!|N(S)|!}{(|S|+|N(S)|)!} f_G(S) - \sum_{\substack{S \in C, v_i \notin S \\ S \cup \{v_i\} \in C}} \frac{|S|!(|N(S)|-1)!}{(|S|+|N(S)|)!} f_G(S),$$

and $N(S)$ *denotes the set of neighbors of coalition S, i.e.,* $N(S) = \bigcup_{v \in S} N(v) \setminus S$.

Proof. Based on the result by Myerson [36], we know that: $MV_i(f_G) = SV_i(f_G^{\mathcal{M}})$. This, as well as Equation (5), imply that:

$$MV_i(f_G) = \frac{1}{|V|!} \sum_{\pi \in \Pi} \left(f_G^{\mathcal{M}}(S_i^{\pi}) - f_G^{\mathcal{M}}(S_i^{\pi} \setminus \{v_i\}) \right). \tag{6}$$

Now, let us consider the marginal contribution of player v_i to coalition S. In particular, based on Equation (1), i.e., the definition of $f_G^{\mathcal{M}}$, the marginal contribution of v_i to S is:

$$f_G^{\mathcal{M}}(S) - f_G^{\mathcal{M}}(S \setminus \{v_i\}) = \sum_{K_j \in \mathcal{K}(S)} f_G(K_j) - \sum_{K_j \in \mathcal{K}(S \setminus \{v_i\})} f_G(K_j).$$

Note that every connected component of S that does not contain v_i appears in both terms of the right-hand side of the equation (see Figure 1 for an illustration). Based on this, it is possible to simplify the equation as follows:

$$f_G^{\mathcal{M}}(S) - f_G^{\mathcal{M}}(S \setminus \{v_i\}) = f_G(K^i(S)) - \sum_{K_j \in \mathcal{K}\left(K^i(S) \setminus \{v_i\}\right)} f_G(K_j), \tag{7}$$

where $K^i(S)$ denotes the component in $\mathcal{K}(S)$ that contains v_i. Based on Equation (7), it is possible to rewrite Equation (6) as follows:

$$MV_i(f_G) = \frac{1}{|V|!} \sum_{\pi \in \Pi} \left(f_G(K^i(S_i^{\pi})) - \sum_{K_j \in \mathcal{K}\left(K^i(S_i^{\pi}) \setminus \{v_i\}\right)} f_G(K_j) \right).$$

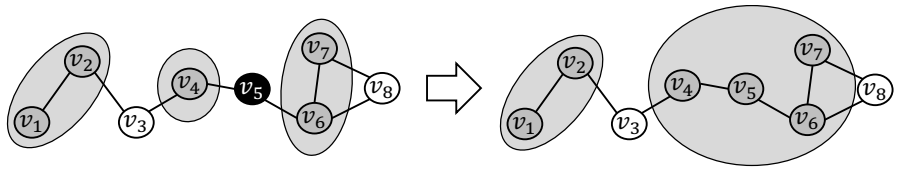


Fig. 1. An illustration of the marginal contribution of player v_5 to coalition $S \setminus \{v_5\} = \{v_1, v_2, v_4, v_6, v_7\}$ in game $f_G^{\mathcal{M}}$ for a sample graph G. From Equation (1), we have: $f_G^{\mathcal{M}}(S) - f_G^{\mathcal{M}}(S \setminus \{v_5\}) = f_G(\{v_1, v_2\}) + f_G(\{v_4, v_5, v_6, v_7\}) - f_G(\{v_1, v_2\}) - f_G(\{v_4\}) - f_G(\{v_6, v_7\}) = f_G(\{v_4, v_5, v_6, v_7\}) - f_G(\{v_4\}) - f_G(\{v_6, v_7\})$.

This, in turn, can be written differently as follows:

$$MV_i(f_G) = \sum_{S \in C, v_i \in S} \frac{|\{\pi \in \Pi : S = K^i(S_i^\pi)\}|}{|V|!} f_G(S) - \sum_{S \in C, v_i \notin S} \frac{|\{\pi \in \Pi : S \in \mathcal{K}(K^i(S_i^\pi) \setminus \{v_i\})\}|}{|V|!} f_G(S).$$

In the first term of the right-hand side of this equation, the condition $S = K^i(S_i^\pi)$ means that S is the component of v_i in S_i^π. Thus, given that $v_i \in S$, this condition is equivalent to the condition: $S \in \mathcal{K}(S_i^\pi)$. In the second term, for every $S \in C : v_i \notin S$, we consider every permutation $\pi \in \Pi$ such that S is one of the components of $S_i^\pi \setminus \{v_i\}$ that are merged into a single component, namely $K^i(S_i^\pi)$, in S_i^π. Thus, v_i must be a neighbor of S. Based on this, we have:

$$MV_i(f_G) = \sum_{S \in C, v_i \in S} \frac{|\{\pi \in \Pi : S \in \mathcal{K}(S_i^\pi)\}|}{|V|!} f_G(S) - \sum_{\substack{S \in C, v_i \notin S \\ S \cup \{v_i\} \in C}} \frac{|\{\pi \in \Pi : S \in \mathcal{K}(S_i^\pi \setminus \{v_i\})\}|}{|V|!} f_G(S). \quad (8)$$

Here, the first fraction is the probability that, in a random permutation π, a coalition $S \in C : v_i \in S$ will be a component in the subgraph induced by S_i^π. This will be the case if and only if: (i) all the players in $S \setminus \{v_i\}$ precede v_i in π; and (ii) all the players in $N(S)$ come after v_i in π. Note that the total number of ways in which the players in $S \cup N(S)$ can be ordered in π equals: $(|S| + |N(S)|)!$. Out of all those orderings, the number of those in which the players in $S \setminus \{v_i\}$ precede v_i and the players in $N(S)$ come after v_i equals: $(|S| - 1)!|N(S)|!$. Based on this, we have:

$$\frac{|\{\pi \in \Pi : S = K^i(S_i^\pi)\}|}{|V|!} = \frac{(|S| - 1)!|N(S)|!}{(|S| + |N(S)|)!}. \quad (9)$$

The second fraction is the probability that, in a random permutation π, a coalition $S \in C : v_i \notin S, S \cup \{v_i\} \in C$ will be a component in the subgraph induced by $S_i^\pi \setminus \{v_i\}$. This will be the case if and only if: (i) all the players in S precede v_i in π; and (ii) all the players in $N(S) \setminus \{v_i\}$ come after v_i in π. The total number of ways in which the players in $S \cup N(S)$ can be ordered in π equals: $(|S| + |N(S)|)!$. Out of all those orderings, the number of those in which the players in S precede v_i and the players in $N(S) \setminus \{v_i\}$ come after v_i equals: $|S|!(|N(S)| - 1)!$. Based on this, we have:

$$\frac{|\{\pi \in \Pi : S \in \mathcal{K}(S_i^\pi \setminus \{v_i\})\}|}{|V|!} = \frac{|S|!(|N(S)| - 1)!}{(|S| + |N(S)|)!}. \quad (10)$$

Equations (8), (9) and (10) imply the correctness of the theorem. □

Based on the above analysis, we construct our algorithm for computing the Myerson value based on our DFSE algorithm for enumerating connected subgraphs. We call this algorithm DFS-Myerson, the pseudocode of which is outlined in Algorithm 3.

New lines compared to DFSE have been highlighted. Here, the parameter *Neighbors* keeps track of the neighbors of S. In the first call of *ExpandSubgraph* (line 7), parameter *Neighbors* is set to be

ALGORITHM 3: The depth-first-search algorithm for calculating the Myerson Value in a Myerson Game (DFS-Myerson)

Input: Graph, $G = (V, E)$; a function, $f_G : C \to \mathbb{R}$, specifying the value of every connected coalition
Output: The Myerson value $MV_i(f_G)$ for every $v_i \in V$

```
1  DFS-Myerson begin
2      Sort V by degree descendingly, and re-index the nodes accordingly;
3      for i ← 1 to |V| do MV_i(f_G) = 0;
4      for i ← 1 to |V| do
5          Sort N(v_i) by degree descendingly, and assign the sorted list to parameter M(v_i);
6      for i ← 1 to |V| do
7          ExpandSubgraph(G, (v_i), S, {v_1, ..., v_{i-1}}, 1, ∅ );
8  ExpandSubgraph(G, path, S, Forbidden, indexOfFirstNeighbor, Neighbors ) begin
9      v ← path.last();
10     for indexOfCurrentNeighbor ← indexOfFirstNeighbor to |M(v)| do
11         u ← M(v).get(indexOfCurrentNeighbor);
12         if (u ∉ S) ∧ (u ∉ Forbidden) then
13             ExpandSubgraph(G, path ∪ {u}, S ∪ {u}, Forbidden, 1, Neighbors );
14             Forbidden ← Forbidden ∪ {u}; Neighbors ← Neighbors ∪ {u};
15         else if u ∈ Forbidden then Neighbors ← Neighbors ∪ {u};
16     path.removeLast();
17     if path.length() > 0 then
18         w ← path.last();
19         indexOfFirstNeighbor ← M(w).getIndex(v) + 1);
20         ExpandSubgraph(G, path, S, Forbidden, indexOfFirstNeighbor, Neighbors );
21     else
22         foreach v_i ∈ S do MV_i(f_G) ← MV_i(f_G) + ((|S|−1)!(|Neighbors|)!)/((|S|+|Neighbors|)!) f_G(S);
23         foreach v_i ∈ Neighbors do MV_i(f_G) ← MV_i(f_G) − ((|S|)!(|Neighbors|−1)!)/((|S|+|Neighbors|)!) f_G(S);
```

the empty set. In the body of the function *ExpandSubgraph* it is modified only in lines 14 and 15. In both cases node u, which is a neighbor of v, is added to the set. Since we proved in Lemma 4.1 (b) that $path \subseteq S$ and from line 9 we know that $v \in path$, parameter $Neighbors$ consists only of neighbors of S. Let us argue that it contains all neighbors of S. From the proof of Lemma 4.1 (e) we know that all neighbors of nodes from S are considered in line 11; therefore, all neighbors of nodes from S which are not in S are added to $Neighbors$.

Finally, lines 22–23 update the Myerson value according to the analysis of the marginal contribution from Theorem 5.1.

6 COMPUTING THE SHAPLEY VALUE IN ARBITRARY CONNECTIVITY GAMES

In this section, we present a new algorithm to compute the Shapley value in an arbitrary connectivity game. As mentioned earlier in the introduction, there already exists an algorithm designed for this

purpose, due to Michalak et al. [33], and our aim is to develop a more efficient alternative based on our enumeration algorithm from Section 4.

As noted by Michalak et al. [33], in order to calculate the Shapley value of a player in a connectivity game, *it suffices to examine only the connected coalitions*, because every non-zero marginal contribution involves a player's addition to, or removal from, a connected coalition. More formally, the authors propose to compute the Shapley value of a player in a connectivity game as follows:[7]

Proposition 6.1 (cf. Michalak et al. [33]). *Given a graph, $G = (V, E)$, and a connectivity game defined by Equation (3)) the following holds for every $v_i \in V$:*

$$SV_i(f_G^{\mathcal{L}}) = \sum_{\substack{S\in C, v_i\in S\\ S\setminus\{v_i\}\in C}} \lambda_S \left(f_G(S) - f_G(S\setminus\{v_i\})\right) + \sum_{\substack{S\in C, v_i\in S\\ S\setminus\{v_i\}\notin C}} \lambda_S\, f_G(S) \;-\; \sum_{\substack{S\in C, v_i\notin S\\ S\cup\{v_i\}\notin C}} \lambda_{S\cup\{v_i\}}\, f_G(S), \quad (11)$$

where $\lambda_S = (|S|-1)!(|V|-|S|)!/(|V|!)$.

Proof. Recall that based on Equations (3) and (4) the Shapley value of player $v_i \in V$ in the connectivity game $(V, f_G^{\mathcal{L}})$ is defined as follows:

$$SV_i(f_G^{\mathcal{L}}) = \sum_{S\subseteq V, v_i\in S} \lambda_S \left(f_G^{\mathcal{L}}(S) - f_G^{\mathcal{L}}(S\setminus\{v_i\})\right).$$

As can be seen in this formula, the coalitions that influence the Shapley value of v_i are: $\{S \subseteq V : v_i \in S\}$. Let us divide this set into four exhaustive and disjoint subsets, depending on whether or not S is connected and whether or not $S \setminus \{v_i\}$ is connected; we end up with the following:

$$SV_i(f_G^{\mathcal{L}}) = \overbrace{\sum_{\substack{S\subseteq V, v_i\in S\\ S\in C, S\setminus\{v_i\}\in C}} \lambda_S\Big(f_G(S)-f_G(S\setminus\{v_i\})\Big)}^{\mathbf{S1}} + \overbrace{\sum_{\substack{S\subseteq V, v_i\in S\\ S\notin C, S\setminus\{v_i\}\in C}} \lambda_S\Big(0-f_G(S\setminus\{v_i\})\Big)}^{\mathbf{S2}}$$
$$+ \underbrace{\sum_{\substack{S\subseteq V, v_i\in S\\ S\in C, S\setminus\{v_i\}\notin C}} \lambda_S\Big(f_G(S)-0\Big)}_{\mathbf{S3}} + \underbrace{\sum_{\substack{S\subseteq V, v_i\in S\\ S\notin C, S\setminus\{v_i\}\notin C}} \lambda_S\Big(0-0\Big)}_{\mathbf{S4}}.$$

See Figure 2 for an illustration. We can disregard the fourth case, **S4**, as it does not influence $SV_i(f_G^{\mathcal{L}})$. The key observations here is that the second term, **S2**, can be computed as follows:

$$\sum_{S\subseteq V, v_i\notin S, S\cup\{v_i\}\notin C, S\in C} \lambda_{S\cup\{v_i\}} \left(0 - f_G(S)\right).$$

In so doing, the term **S2** can be computed by iterating over connected coalitions, rather than disconnected ones. The Shapley value formula can then be simplified as follows:

$$SV_i(f_G^{\mathcal{L}}) = \sum_{\substack{S\subseteq V, v_i\in S\\ S\in C, S\setminus\{v_i\}\in C}} \lambda_S \left(f_G(S)-f_G(S\setminus\{v_i\})\right) + \sum_{\substack{S\subseteq V, v_i\in S\\ S\in C, S\setminus\{v_i\}\notin C}} \lambda_S\, f_G(S) \;-\; \sum_{\substack{S\subseteq V, v_i\notin S\\ S\cup\{v_i\}\notin C, S\in C}} \lambda_{S\cup\{v_i\}}\, f_G(S).$$

Consequently, we can now traverse only connected coalitions and obtain Equation (11). □

The above analysis shows that, in order to compute the Shapley value in connectivity games, it is crucial to not only enumerate all connected subgraphs, but also identify the cut vertices of each enumerated subgraph. The challenge is then to efficiently combine the enumeration of

[7] The decomposition in Proposition 6.1 was already used in [33] but without a formal proof.

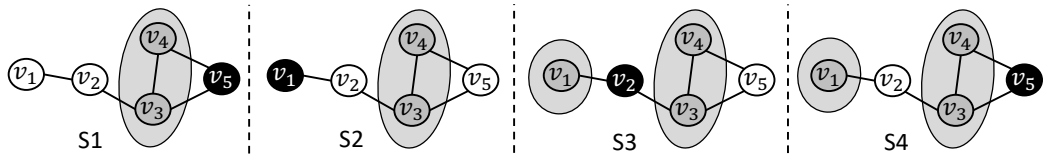


Fig. 2. An illustration of different types of the marginal contribution in game $f_G^{\mathcal{L}}$ for a sample graph G: the marginal contribution of v_5 to coalition $\{v_3, v_4\}$ equals $f_G(\{v_3, v_4, v_5\}) - f_G(\{v_3, v_4\})$ (case **S1**), and of v_1 equals $0 - f_G(\{v_3, v_4\})$ (case **S2**); the marginal contribution of v_2 to coalition $\{v_1, v_3, v_4\}$ equals $f_G(\{v_1, v_2, v_3, v_4\}) - 0$ (case **S3**), and of v_5 equals $0 - 0$ (case **S4**).

induced connected subgraphs with the identification of cut vertices. To this end, Michalak et al. [33] combined Moerkotte and Neumann's algorithm for enumerating connected induced subgraphs with algorithm for identifying cut vertices by Hopcroft and Tarjan [18]. As we mentioned earlier in the introduction, the former algorithm is based on breadth-first search, whereas the latter is based on depth-first search. As such, due to their orthogonal design principles, the two algorithms are hard to combine without a computational overhead; see the work [33] for more details.

As opposed to Moerkotte and Neumann's algorithm for enumerating connected induced subgraphs, which uses breadth-first search, our DFSE algorithm uses depth-first search, making it compatible with algorithm for identifying cut vertices by Hopcroft and Tarjan [18]. More specifically, by using our depth-first-search enumeration, the edges that are used to enlarge the subgraph are guaranteed to form a *Trémaux tree*—an important structure in graph theory, which is used in Hopcroft and Tarjan's algorithm. In more detail, a Trémaux tree, T, of a graph, G, is a rooted spanning tree—a subgraph consisting of all nodes and a subset of edges, which forms a tree rooted at one of its nodes—which satisfies the following property: for any two nodes that have an edge between them in G, one of them is an ancestor of the other in T; see the appendix for an illustrated example.[8] Now, let us show how this property of a Trémaux tree facilitates the identification of cut vertices in a subgraph. To this end, let v be an arbitrary node in G, and consider a subtree, S, rooted at a child of v in T. The removal of v from G disconnects nodes from S if and only if there is no edge in G that connects S to other parts of the graph. From the property of Trémaux tree, all such potential edges would go to the ancestors of v (we will call them *backedges*). Thus, to identify cut vertices, it suffices to know the node nearest to the root that can be reached from the children's subtrees. This information can be easily updated recursively when we backtrack in a depth-first search—for the subtree rooted at v, it is one of the nodes connected to v or one of the nearest nodes that can be reached from its subtrees.

The pseudocode is presented in Algorithm 4. New lines compared to DFSE have been highlighted. To gather extra information, we expand the recursive function from Algorithm 1 by three new parameters. The parameter *Neighbors* keeps track of the neighbors of S in the same way as in Algorithm 3—see the description therein for details. Let us discuss the other two new parameters. Assume that the root of a tree is on level 1, and its children are on level 2, and so on. Now, for each node, v, at the position k on the *path*, the position k on the list *low* keeps track of the level of the node nearest to the root that can be reached from v or the children's subtrees; we will denote this value $low(v)$. When a new node is added to the *path* (lines 8 and 14) $low(v)$ is initialized to infinity. Then, it is updated in two situations. First, when an edge from v to its ancestor is analyzed—if this ancestor, u, is closer to the root than the previously found (stored on the list *low*), then $low(v)$ on

[8]Note that breath-first-search enumeration does not create a Trémaux tree. In result, it is not possible create a breadth-first search algorithm to identify cut-vertices to combine it with Moerkotte and Neumann's algorithm.

ALGORITHM 4: The depth-first-search algorithm for calculating the Shapley Value in a connectivity game (DFS-Shapley)

Input: Graph $G = (V, E)$; characteristic function for the connectivity game $f_G^{\mathcal{L}}$
Output: The Shapley value $SV_i(f_G^{\mathcal{L}})$ for every $v_i \in V$

```
1  DFS-Shapley(G) begin
2      Sort V by degree descendingly, and re-index the nodes accordingly;
3      for i ← 1 to |V| do SV_i(f_G^L) = 0;
4      for i ← 1 to |V| do
5          Sort N(v_i) by degree descendingly, and assign the sorted list to parameter M(v_i);
6      for i ← 1 to |V| do
7          S ← {v_i};
8          ExpandSubgraph(G, (v_i), S, {v_1, ..., v_{i-1}}, 1, ∅, (∞), ∅ );
9  ExpandSubgraph(G, path, S, Forbidden, indexOfFirstNeighbor, Neighbors, low, SC ) begin
10     v ← path.last(); l ← low.last();
11     for indexOfCurrentNeighbor ← indexOfFirstNeighbor to |M(v)| do
12         u ← M(v).get(indexOfCurrentNeighbor);
13         if (u ∉ S) ∧ (u ∉ Forbidden) then
14             ExpandSubgraph(G, path ∪ {u}, S ∪ {u}, Forbidden, 1, Neighbors, low ∪ {∞}, SC );
15             Forbidden ← Forbidden ∪ {u}; Neighbors ← Neighbors ∪ {u};
16         else if u ∈ Forbidden then Neighbors ← Neighbors ∪ {u};
17         else if path.getIndex(u) < low.last() then
18             l ← path.getIndex(u);
19             low.updateLast(l);
20     path.removeLast(); low.removeLast();
21     if path.length() > 0 then
22         w ← path.last();
23         if l ≥ path.length() then SC.add(w);
24         else if l < low.last() then low.updateLast(l);
25         indexOfFirstNeighbor ← M(w).getIndex(v) + 1;
26         ExpandSubgraph(G, path, S, Forbidden, indexOfFirstNeighbor, Neighbors, low, SC );
27     else
28         if v was added only once SC then SC.remove(v);
29         foreach v_i ∈ SC do SV_i(f_G^L) ← SV_i(f_G^L) + λ_S(f_G(S));
30         foreach v_i ∈ S \ SC do SV_i(f_G^L) ← SV_i(f_G^L) + λ_S(f_G(S) − f_G(S \ {v_i}));
31         foreach v_i ∈ V \ (S ∪ Neighbors) do SV_i(f_G^L) ← SV_i(f_G^L) − λ_{S∪{v_i}}(f_G(S));
```

the list is updated to the level of u (lines 17–19). Second, when we backtrack from a child. This case is handled in lines 20–26. In line 20, node v is removed from the *path*, and value $low(v)$ is

removed from the *low*, being earlier assigned to variable l. Since we backtrack from v, l contains the correct value of $low(v)$, as all its children and backedges have been already analyzed. Now, let w be the last node on the *path* after this operation (in fact, this node is indeed assigned to w later on in line 22). Then, $low.last()$ keeps track of $low(w)$, i.e., is equal to the level of the node nearest to the root—from the nodes found so far—that can be reached from w or the children's subtrees. If l is smaller than $low.last()$, i.e., $low(v) < low(w)$, it means that a node nearer to the root than the previously found can be reached using the v's subtree. That is why $low.last()$ is updated (line 24).

Let us discuss the third and the last parameter: *SC*. Set *SC* contains all cut vertices from *S*. As discussed earlier, a node, w, is a cut vertex if and only if it has a child, v, for whom the node nearest to the root that can be reached from this child or the children's subtrees is w. Using our notation, this happens if and only if $low(v)$ is equal to the level of w. In the pseudocode, having been initialized to the empty set in line 8, *SC* is modified only when the algorithm backtracks in lines 20–26: if w, v are the last two nodes on the path (in that order) in line 20, then w is a parent of v, from which we backtrack. If $low(v)$, stored in l, is larger or equal to the level of w, equal to the position of w on the path, i.e., length of the path in line 23, then w is a cut vertex and it is added to *SC*. This reasoning applies to all nodes which are not the root of the Tremaux tree, i.e., the first node on the path. This is because, for all such nodes, removing w would disconnects node v from the rest of the subgraph. However, the root is added to the set *SC* for every child, and it is a cut vertex, if and only if it has more than one child in the Tremaux tree. That is why, in line 28 we remove the root from *SC* if it was added only once. Finally, for every enumerated induced connected subgraph, in lines 29–31 the Shapley value is updated based on sets $Neighbors$ and *SC*, according to the Proposition 6.1.

Compared to our DFSE algorithm from Section 4.1, the asymptotic time of our DFS-Shapley algorithm has not changed with respect to the enumeration of connected subgraphs, i.e., it is $O(|C||E|)$.[9]

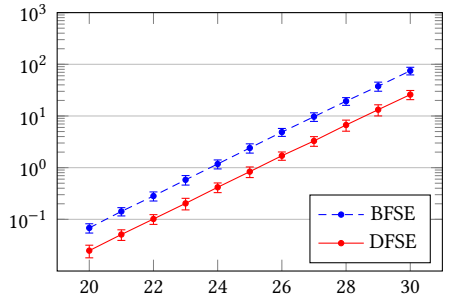
7 EMPIRICAL EVALUATION

We test the performance of our algorithms on three types of randomly-generated networks, typically studied in the complex-networks literature as they cover a wide spectrum of structural features of real-life networks [30, 35], namely:

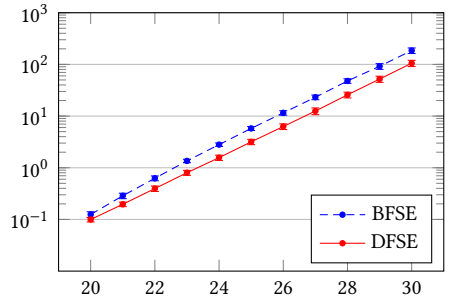
(1) *Scale-free* networks generated using the Barabási-Albert model [5]. These networks, built gradually, a node after a node, are parametrised with two parameters, (n, k), where n is the total number of nodes, and k is the number of links added with each node;
(2) *Small-world* networks, generated using the Watts-Strogatz model [54]. These networks are parametrised with three parameters, (n, d, r), where n is the number of nodes, d is the average degree, and r is the rewiring probability;
(3) *Random graphs* generated using the Erdős-Rényi model [13]. These networks, in which each edge exists with a certain probability, independently of other edges, are parametrised with two parameters, (n, d), where n is the number of nodes, and d is the expected average degree.

For each type of randomly-generated networks, we run the algorithms over 100 such networks. All experiments are carried out on 3,5 GHz Intel Core i5 with 16 GB 2400 MHz DDR4 RAM.

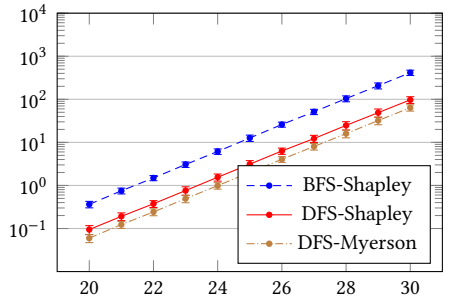
[9]Note, that to obtain a constant time of $path.getIndex(u)$ operation in line 17, the parameter $path$ should be enriched with an associative array or alternatively additional array of nodes' levels can be passed along.



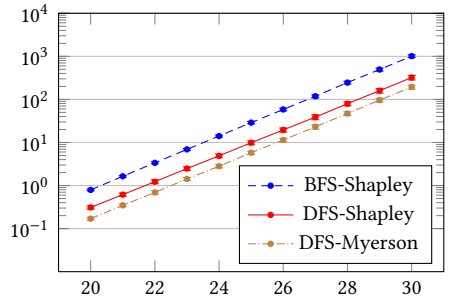
(a) Scale-free graphs (the number of edges added with each node is 4).



(b) Erdős-Rényi graphs (the expected average degree is 4).



(c) Scale-free graphs (the number of edges added with each node is 4).



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

Fig. 3. For different numbers of players (the x-axis), the figures depict the runtime of the algorithms in seconds (the y-axis). Pseudocodes of our algorithms—DFSE, DFS-Myerson and DFS-Shapley—can be found in Algorithms 1, 3, and 4. BFSE was proposed by Moerkotte and Neumann [34]. BFS-Shapley was proposed by Michalak et al. [33].

Our experiments show that the DFSE algorithm outperforms the BFSE one. In particular, Figure 3 (a) depicts the running time for sparse scale-free graphs, typically used to model contact networks (see the work by Leary et al. [25] for examples). As can be seen, as n increases, the ratio of both algorithms does not change and oscillates between 2.7 and 3.0. For instance, for $n = 30$, our DFSE algorithm takes on average 25 seconds, while it takes 74 seconds for BFSE enumeration to finish. Figure 3 (b) depicts the running time for more dense random Erdős-Rényi graphs. For smaller graphs DFSE is only marginally faster than BFSE. However, for larger graphs ($n > 23$) the difference is visible and exceeds 1.7.

In Figure 3 (c) and (d) we compare the performance of our new algorithm—DFS-Shapley—for calculating the Shapley value with the algorithm proposed by Michalak et al. [33], based on BFSE and denoted by BFS-Shapley. As we can see, our DFS-Shapley algorithm is 4 times faster in sparse graphs (part (c)) and 3 times faster in dense graphs (part (d)). Importantly, only part of the advantage comes from the faster enumeration algorithm. Moreover, the advantage increases very slightly with the number of nodes. This can be justified as follows: although the pessimistic number of steps for every found connected subgraph in both enumeration algorithms equals $O(|E|)$, for sparse graphs, this number does not exceed a small constant. Thus, searching separately for the cut vertices, which is a part of BFS-Shapley, is an additional action which takes linear time in respect to the number

Network	$\|V\|$	$\|E\|$	$\|C\|$	DFSE	BFSE	DFS-Myerson	DFS-Shapley	BFS-Shapley
Women [10]	18	139	0.2M	0.08	0.51	0.15	0.20	1.65
CEOs [15]	26	284	67M	17.56	47.95	26.91	50.90	161.54
WTC [23]	36	64	401M	12.17	36.11	41.89	65.52	320.84
Karate [56]	34	78	3923M	132.88	533.38	390.01	681.68	3065.77

Table 1. The runtime (in seconds) of DFSE, BFSE, DFS-Myerson, DFS-Shapley, and BFS-Shapley algorithms for small real-life social network datasets from the literature.

of edges, i.e., $O(|E|)$. On the other hand, our new algorithm performs only $O(|V|)$ steps updating Shapley value for all the nodes. For small-world graphs results are comparable to Erdős-Rényi graphs and can be found in the appendix.

We also report the performance of the DFS-Myerson algorithm. Note that since this algorithms is the first general algorithm for computing the Myerson value, there is no algorithm to compare it with. We observe that DFS-Myerson is slower than DFSE, that only enumerates induced connected subgraphs, as for each found subgraph it has to update the Shapley value for all nodes. However, as expected earlier, it is only slightly faster than DFS-Shapley, that requires finding pivotal nodes. This is because additional structures (low, SC) need to be passed recursively which slightly increases the runtime of the algorithm.

Table 1 shows the running time of all five algorithms for various small real-life social networks from the literature. Results are consistent with the experiments for random networks. Recall that the pessimistic time complexity of all algorithms is $O(|C||E|)$. Thus, the algorithms performance depends not only on the number of nodes and edges in the graph, but also on the number of induced connected subgraphs, $|C|$, as can be seen in Table 1.

A more extensive empirical evaluation of the algorithms introduced in this article can be found in the appendix.

8 THE MYERSON AND SHAPLEY VALUES AS CENTRALITY MEASURES—A COMPARISON BASED ON THE APPLICATION TO TERRORIST NETWORKS

In this section, we present a sample application of the algorithms. In particular, we compare the performance of the Shapley value and the Myerson value as the tools to identify key figures in terrorist networks. The possibility to apply social network analysis techniques to investigate criminal and terrorist networks has raised much interest in the literature [40]. A particular attention has been paid to the problem of identifying key terrorists [24, 32, 42]. This not only helps to understand the hierarchy within these organizations but also allows for a more efficient deployment of scarce investigation resources.

One possible approach to the above problem is to infer the importance of different individuals using centrality analysis, i.e., from the topology of the covert network [23, 51]. However, Lindelauf et al. [26] argued that classic centrality measures are inadequate in the context of terrorist networks and proposed to use the Shapley value in connectivity games instead. The aim was to develop a more sophisticated, parametrised centrality measure that delivers new insights when compared to classic measures. Unfortunately, with no existing dedicated algorithm for the Shapley value in connectivity games, the analysis presented by Lindelauf et al. [26] was limited only to few specifications of the connectivity game (i.e. the forms of the value function). However, using the algorithms proposed in this article, we are able to analyse a much wider spectrum of the value functions (Section 8.1).

Furthermore, we are also able to compare the Shapley value-based centrality measure to a measure based on the Myerson value (Section 8.2).

In what follows, we will often refer to "the Shapley value in connectivity games (by Amer and Giménez [2])" as simply "the Shapley value".

8.1 The Analysis of the Shapley Value/Connectivity Game-Based Centrality for Terrorist Networks

Lindelauf et al. [26] seeks the centrality measure for terrorist networks that takes into account the following two factors:

- a *"connectivity" factor*—the measure should be able to quantify the role of individual terrorists in connecting various groups of nodes within the network; and
- an *"extra-information" factor*—the mesure should be able to take into account any additional intelligence available about individual terrorists (such as experience, 'combat training, etc.).

The authors argued that the Shapley value for connectivity game $f_G^{\mathcal{L}}$, as defined by formula (3), accounts for both these factors. As for the "connectivity" factor, $f_G^{\mathcal{L}}$ assigns high marginal contributions to cut vertices in a given coalition. As a result, the nodes which are the cut vertices more often than others will have the relatively higher Shapley values (hence, they will be relatively higher in the ranking of nodes). As for the second factor, one can manipulate function $f_G(S)$ in formula (3) to incorporate any available additional information about the nodes and links between them. In particular, Lindelauf et al. studied the following alternative definitions of $f_G(S)$, where ω_{ij} and ω_i denote weights of edges and nodes, respectively:

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

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).

In the remainder of this section, we use Algorithm 4 to study a wider variety of f_G functions. In particular, we propose the following generalized form of f_G:

$$f_G(S) = (|S|)^{\alpha} \times (|E(S)|)^{\beta} \times \left(\sum_{v_i\in S}\frac{\omega_i}{|S|}\right)^{\gamma} \times \left(\sum_{\{v_i,v_j\}\in E(S)}\frac{\omega_{ij}}{|E(S)|}\right)^{\delta}, \tag{13}$$

where $\alpha, \beta, \gamma, \delta \in \mathbb{R}$ are parameters for exponents. We note that the third and fourth components of the above product are average weights of node and edge in the subgraph, respectively. By varying those parameters we can obtain many different forms of f_G, including functions (12) (i) and (12) (ii) analyzed by Lindelauf *et al.*[10] Naturally, in the simplest form, when all parameters equal zero, then formula (13) becomes the generic connectivity game by Amer and Giménez, i.e. $f_G(S) = 1$.

In our simulations, we set α and β to be integer values between -2 and 2 and we impose the additional condition that $\alpha + \beta \geq 0$ in order to preserve monotonicity. Now, the average weights of nodes and of edges, if appear, either multiply or divide the product of the first two components. Thus, $\gamma, \delta \in \{-1, 0, +1\}$. Overall, we study 135 different definitions of f_G. For each configuration of parameters, we calculate the ranking of nodes obtained using formula (13) and compare it with

[10]Function (12) (i) is obtained by setting $\delta = -1$ and $\alpha = \beta = \gamma = 0$ and function (12) (ii) by setting $\alpha = \gamma = 1$ and $\beta = \delta = 0$.

$\alpha\backslash\beta$		2	1	0	-1	-2
	$\gamma\backslash\delta$	1|0|-1	1|0|-1	1|0|-1	1|0|-1	1|0|-1
2	1|0|-1	6.0	6.0	5.9	5.9	4.9
1	1|0|-1	6.0	6.0	6.0	5.1	—
0	1|0|-1	6.0	6.0	5.7	—	—
-1	1|0|-1	6.0	6.0	—	—	—
-2	1|0|-1	6.0	—	—	—	—

(a) The average intersection of rankings

2	1	0	-1	-2
1|0|-1	1|0|-1	1|0|-1	1|0|-1	1|0|-1
1.3-1.3	1.3-1.3	1.3-1.3	0.7-1.3	1.3-2.5
1.3-1.3	1.3-1.3	1.0-1.3	0.8-2.0	—
1.3-1.3	1.0-1.3	0.0-1.0	—	—
1.3-1.3	0.3-1.3	—	—	—
0.7-1.3	—	—	—	—

(b) The average distance in positions

Table 2. The Shapley value/connectivity game-based approach. The comparison of the top 6 terrorists returned by the parametrized characteristic function from formula (13) with the top 6 terrorists returned by the simple 0-1-connectivity game in formula (2).

the ranking for the 0-1-connectivity game in formula (2). Since the main goal of this exercise is to identify key players, in our comparison we focus on the $\sqrt{|V|} = 6$ terrorists that were ranked highest by both methods.

With Algorithm 4, we are able to work with the bigger version of the terrorist network responsible for the 9/11 World Trade Center attacks [23], composed of 36 nodes and 64 edges.[11] In Table 2, we report the following results of the simulations:

- In each cell of Table 2 (a), we present the average size of the intersection of the top 6 terrorists in both rankings, i.e. in the ranking obtained from formula (13) and in the ranking obtained from the 0-1-connectivity game. For instance, the cell in the third row and the third column shows the average intersection of rankings for $\alpha = \beta = 2$ and nine combinations of γ and δ, i.e., $(1, 1)$, $(1, 0)$, $(1, -1)$, $(0, 1)$, $(0, 0)$, $(0, -1)$, $(-1, 1)$, $(-1, 0)$, and $(-1, -1)$. The average intersection in this particular cell is 6 which says that, for all the nine combinations of parameters, formula (13) returned exactly the same ranking as the 0-1-connectivity game in formula (2). In fact, we observe that the top 6 members have changed only in 15% of the tests. In other words, in 85% of cases, the top 6 nodes returned by the parametrised formula (13) are exactly the same as the top 6 nodes returned by the simple 0-1-connectivity game in formula (2). Furthermore, in only about 2% of the cases, both ranking differed with more than 1 terrorist.
- To examine the differences between the positions of top 6 terrorists in both rankings,[12] in Table 2 (b), we calculate the average distance between the positions of top 6 terrorists in both rankings. Here, each cell presents the maximum and minimum value of this average. We observe that, not only the top 6 terrorist rarely differ, but the change of their positions in the rankings are usually minor. In more detail, the average change of position rarely exceeded 1.3.

The above results suggest that, for such networks as the WTC one, the choice of $f_G(S)$ essentially does not matter. The main reason behind this phenomenon appears to be the fact that $f_G^{\mathcal{L}}$ assigns relatively very high contributions to pivotal players, and only incremental marginal contributions to non-pivotal players. This seems to be magnified by the fact that we deal here with a sparse network. Specifically, in this network, out of all 6 billions induced subgraphs, only 0.6% are connected.

[11] Note that the 19 node version of this network studied by Lindelauf et al. [26] was composed only the planes' hijackers. The bigger network studied in this article contains also many other individuals involved in the preparation of the attack.
[12] Observe that their positions could differ although the rankings were composed of the same 6 terrorists.

$\alpha\backslash\beta$		2	1	0	-1	-2	2	1	0	-1	-2
	$\gamma\backslash\delta$	1|0|-1	1|0|-1	1|0|-1	1|0|-1	1|0|-1	1|0|-1	1|0|-1	1|0|-1	1|0|-1	1|0|-1
2	1|0|-1	4.8	4.7	4.3	1.1	0.0	1.2-2.3	1.0-2.2	0.8-4.2	7-24	24-28
1	1|0|-1	5.0	4.9	1.1	0.0	—	1.2-2.8	1.0-2.8	8.8-28	21-28	—
0	1|0|-1	4.4	4.2	0.0	—	—	1.7-3.7	2.5-8.2	24-30	—	—
-1	1|0|-1	3.3	0.8	—	—	—	2.0-7.0	13-26	—	—	—
-2	1|0|-1	1.1	—	—	—	—	11-22	—	—	—	—

(a) The average intersection of rankings (b) The average distance in positions

Table 3. The Myerson value-based approach. The comparison of the top 6 terrorists returned by the parametrized characteristic function from formula (1) with the top 6 terrorists returned by the simple 0-1-connectivity game in formula (2).

Furthermore, the average number of pivotal players in a connected subgraph is high (8 to be precise). Such nodes that are often crucial from the connectivity point of view will have a high ranking as they will be awarded positive marginal contributions (that equal to the entire value of the coalition), irrespective of the form of $f_G^{\mathcal{L}}$.

To summarise our analysis of the Shapley value in connectivity games used as a centrality measure, we conclude that this approach mainly focuses on the role of nodes in connecting the network. The exact values of the characteristic function will play the secondary, if not negligible, role. In our sample terrorist network application, it means that the result is most often driven by the "connectivity" factor and, in such cases, the "extra information" factor plays a negligible role.

8.2 The Analysis of the Myerson Value Centrality for Terrorist Networks

Let us now perform a similar analysis to the one in the previous section, but now using the Myerson value as the measure of centrality for terrorist networks. The results are presented in Table 3 in the same way as in Table 2. We observe that this measure is more sensitive to changes of $f_G(S)$ than the Shapley value. In particular, the average intersection of the top 6 terrorists in both rankings—i.e., in the ranking obtained from formula (13) and in the ranking from the 0-1-connectivity game—is much lower in Table 2 (b) for the Myerson value than in Table 2 (a) for the Shapley value. This means that, when we modify the characteristic function, the ranking typically changes—the result that we have been looking for. This if further confirmed by the average distances between the positions of top 6 terrorists in both rankings—they are now much higher than in Table 2 (b), i.e. the positons of top 6 terrorists significantly differ, on average.

Our results suggest that the Myerson value does not favour pivotal players as much as the Shapley value, i.e. the "connectivity" factor becomes relatively less dominant in the evaluation of the nodes. This means that the Myerson value and not the Shapley value seems to be better suited to analyse covert networks in which the "extra-information" factor (and not only the "connectivity" factor) should be taken into account On the other hand, the Shapley value may be used in dense network—to highlight the hidden "connectivity" factor—or in application where "extra-information" are not important or available. Also, we refer the reader to the recent work that suggest that, in general, centralities based on the coalitional game theory are good choice when the nodes are assessed based on some property that adheres to Myerson's notion of Fairness [46].

9 CONCLUSIONS

In this article, we developed a depth-first search algorithm, called DFSE, for enumerating all connected induced subgraphs in any given graph. Building upon DFSE, we developed two additional algorithms. The first, called DFS-Myerson, computes the Myerson value in a Myerson game, whereas the second algorithm, called DFS-Shapley, computes the Shapley value in a connectivity game. Our empirical evaluation showed that DFSE outperforms the previous state-of the-art algorithm in the literature due to Moerkotte and Neumann [34] based on the breadth first search. Likewise, our empirical evaluation showed that DFS-Shapley outperforms the only available alternative in the literature, due to Michalak et al. [33]. As for our DFS-Myerson algorithm, no other alternative exists in the literature to date.

We hope that our algorithm will make it possible to apply the Myerson value to new more complex applications and verify its usefulness. However, an important limitation of the exact computation of the Myerson and Shapley values in arbitrary graph-restricted games is the exponential time complexity, caused by the exponential input size. To address this limitation, in our future work, we plan to consider approximation algorithms and further study the computational properties of these values but for specific characteristic functions and classes of graphs. For instance, it is interesting to develop a centrality measure dedicated to covert networks analysis that is based on the Myerson or the Shapley value but that can be computed in the polynomial time.

ACKNOWLEDGMENTS

This article is an extended version of a paper originally presented at AAMAS-14 [47], where a number of new technical results have been added. Specifically, in this version all the properties of the DFSE algorithm—soundness, completeness, non-redundancy and computational complexity—have been considered and formally proven in Theorems 4.2–4.4. Furthermore, we have added the proofs of Propositions 4.5 and 4.6. We have also formalized and proven the closed-form formulas for the Myerson and Shapley values in Theorem 5.1 and Proposition 6.1. Finally, we described in more details related work (Section 2) and significantly extended the experimental and application sections (Sections 7 and 8).

Oskar Skibski and Tomasz Michalak were supported by the Polish National Science Centre grant 2015/19/D/ST6/03113. Tomasz Michalak and Michael Wooldridge were supported by the European Research Council under Advanced Grant 291528 ("RACE"). Earlier version of this work was also supported by the Polish National Science Centre grant 2013/09/D/ST6/03920.

REFERENCES

[1] Encarnacion Algaba, Jesús M. Bilbao, Julio R. Fernández, Nieves Jiménez, and Jorge J. López. 2007. Algorithms for computing the Myerson value by dividends. In *Discrete Mathematics Research Progress*. Nova Science Publishers, Chapter 6, 126–137.

[2] Rafael Amer and José Miguel Giménez. 2004. A connectivity game for graphs. *Mathematical Methods of Operations Research* 60, 3 (2004), 453–470. Issue 3.

[3] Rafael Amer, José Miguel Giménez, and Antonio Magaña. 2007. Accessibility in oriented networks. *European Journal of Operational Research* 180, 2 (2007), 700–712.

[4] David Avis and Komei Fukuda. 1996. Reverse Search Enumeration. *Discrete Applied Mathematics* 65 (1996), 21–46.

[5] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *Science* 286, 5439 (1999), 509–512.

[6] Jesus-Mario Bilbao. 1998. Values and potential of games with cooperation structure. *International Journal of Game Theory* 27, 1 (1998), 131–145.

[7] Filippo Bistaffa, Alessandro Farinelli, Jesús Cerquides, Juan Rodríguez-Aguilar, and Sarvapali D. Ramchurn. 2017. Algorithms for Graph-Constrained Coalition Formation in the Real World. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 4 (2017), 60.

[8] Adrian Brüngger, Ambros Marzetta, Komei Fukuda, and Jurg Nievergelt. 1999. The parallel search bench ZRAM and its applications. *Annals of Operations Research* 90 (1999), 45–63.

[9] Georgios Chalkiadakis, Edith Elkind, and Michael Wooldridge. 2011. Computational aspects of cooperative game theory. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 5, 6 (2011), 1–168.

[10] Allison Davis, Burleigh Bradford Gardner, and Mary R Gardner. 2009. *Deep South: A social anthropological study of caste and class.* Univ of South Carolina Press.

[11] Mónica del Pozo, Conrado Manuel, Enrique González-Arangüena, and Guillermo Owen. 2011. Centrality in directed social networks. A game theoretic approach. *Social Networks* 33, 3 (2011), 191–200.

[12] Edith Elkind. 2014. Coalitional games on sparse social networks. In *Proceedings of the 10th Conference on Web and Internet Economics (WINE)*. Springer, 308–321.

[13] Paul Erdős and Alfréd Rényi. 1959. Some further statistical properties of the digits in Cantor's series. *Acta Mathematica Hungarica* 10, 1-2 (1959), 21–29.

[14] Julio R. Fernández, Encarnacion Algaba, Jesús M. Bilbao, A. Jiménez, Nieves Jiménez, and Jorge J. López. 2002. Generating functions for computing the Myerson value. *Annals of Operations Research* 109, 1-4 (2002), 143–158.

[15] Joseph Galaskiewicz. 2016. *Social organization of an urban grants economy: A study of business philanthropy and nonprofit organizations.* Elsevier.

[16] Daniel Gómez, Enrique González-Arangüena, Conrado Manuel, Guillermo Owen, Mónica del Pozo, and Juan Tejada. 2003. Centrality and power in social networks: A game theoretic approach. *Mathematical Social Sciences* 46, 1 (2003), 27–54.

[17] Daniel Gómez, Enrique González-Arangüena, Conrado Manuel, Guillermo Owen, Mónica del Pozo, and Juan Tejada. 2004. Splitting graphs when calculating Myerson value for pure overhead games. *Mathematical Methods of Operations Research* 59, 3 (2004), 479–489.

[18] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM* 16, 6 (1973), 372–378.

[19] Bryan Horling and Victor Lesser. 2004. A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review* 19, 04 (2004), 281–316.

[20] Matthew O. Jackson. 2005. A survey of network formation models: stability and efficiency. In *Group Formation in Economics: Networks, Clubs, and Coalitions*. Cambridge University Press: Cambridge, MA, USA, 11–49.

[21] Rakpong Kaewpuang, Dusit Niyato, Puay-Siew Tan, and Ping Wang. 2017. Cooperative Management in Full-Truckload and Less-Than-Truckload Vehicle System. *IEEE Transactions on Vehicular Technology* 66, 7 (2017), 5707–5722.

[22] Shant Karakashian, Berthe Y. Choueiry, and Stephen G. Hartke. 2013. *An algorithm for generating all connected subgraphs with k vertices of a graph.* Technical Report. University of Nebraska-Lincoln.

[23] Valdis E. Krebs. 2002. Mapping networks of terrorist cells. *Connections* 24 (2002), 43–52. Issue 3.

[24] Vito Latora and Massimo Marchiori. 2004. How the science of complex networks can help developing strategies against terrorism. *Chaos, solitons & fractals* 20, 1 (2004), 69–75.

[25] Christopher C. Leary, Markus Schwehm, Martin Eichner, and Hans-Peter Duerr. 2007. Tuning degree distributions: Departing from scale-free networks. *Physica A: Statistical Mechanics and its Applications* 382, 2 (2007), 731–738.

[26] Roy Lindelauf, Herbert Hamers, and Bart Husslage. 2013. Cooperative game theoretic centrality analysis of terrorist networks: The cases of Jemaah Islamiyah and Al Qaeda. *European Journal of Operational Research* 229, 1 (2013), 230–238.

[27] Michael Maschler, Eilon Solan, and Shmuel Zamir. 2013. *Game Theory.* Cambridge University Press.

[28] Sean Maxwell, Mark R. Chance, and Mehmet Koyutürk. 2014. Efficiently enumerating all connected induced subgraphs of a large molecular network. In *International Conference on Algorithms for Computational Biology*. Springer, 171–182.

[29] Reshef Meir, Yair Zick, Edith Elkind, and Jeffrey S. Rosenschein. 2013. Bounding the cost of stability in games over interaction networks. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, 690–696.

[30] Simachew A. Mengiste, Ad Aertsen, and Arvind Kumar. 2015. Effect of edge pruning on structural controllability and observability of complex networks. *Scientific reports* 5 (2015), 18145.

[31] Tomasz P. Michalak, Karthik V. Aadithya, Piotr L. Szczepański, Balaraman Ravindran, and Nicholas R. Jennings. 2013. Efficient computation of the Shapley value for game-theoretic network centrality. *Journal of Artificial Intelligence Research* 46 (2013), 607–650.

[32] Tomasz P. Michalak, Talal Rahwan, Oskar Skibski, and Michael Wooldridge. 2015. Defeating terrorist networks with game theory. *IEEE Intelligent Systems* 30, 1 (Jan. 2015), 53–61.

[33] Tomasz P. Michalak, Talal Rahwan, Piotr L. Szczepański, Oskar Skibski, Ramasuri Narayanam, Michael Wooldridge, and Nicholas R. Jennings. 2013. Computational analysis of connectivity games with applications to the investigation

of terrorist networks. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI Press, 293–301.
[34] Guido Moerkotte and Thomas Neumann. 2006. Analysis of two existing and one new dynamic programming algorithm for generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 930–941.
[35] Mohsen Mosleh and Babak Heydari. 2017. Fair topologies: community structures and network hubs drive emergence of fairness norms. *Scientific reports* 7, 1 (2017), 2686.
[36] Roger B. Myerson. 1977. Graphs and cooperation in games. *Mathematical Methods of Operations Research* 2, 3 (1977), 225–229.
[37] Stefan Napel, Andreas Nohn, and José Maria Alonso-Meijide. 2012. Monotonicity of power in weighted voting games with restricted communication. *Mathematical Social Sciences* 64, 3 (2012), 247–257.
[38] Sonja Nikolić, Goran Kovačević, Ante Miličević, and Nenad Trinajstić. 2003. The Zagreb indices 30 years after. *Croatia Chemica Acta* 76 (2003), 113–124.
[39] Andrzej S. Nowak and Tadeusz Radzik. 1994. The Shapley value for n-person games in generalized characteristic function form. *Games and Economic Behavior* 6, 1 (1994), 150–161.
[40] Steve Ressler. 2006. Social network analysis as an approach to combat terrorism: past, present and future research. *Homeland Security Affairs* 2 (2006), 1–10.
[41] Walid Saad, Zhu Han, Mérouane Debbah, Are Hjørungnes, and Tamer Başar. 2009. Coalitional game theory for communication networks. *Signal Processing Magazine, IEEE* 26, 5 (2009), 77–97.
[42] Marc Sageman. 2004. *Understanding terror networks*. University of Pennsylvania Press.
[43] Estela Sánchez and Gustavo Bergantiños. 1999. Coalitional values and generalized characteristic functions. *Mathematical Methods of Operations Research* 49, 3 (1999), 413–433.
[44] Lloyd S. Shapley. 1953. A value for n-person games. In *Contributions to the Theory of Games*, H.W. Kuhn and A.W. Tucker (Eds.). Vol. II. Princeton University Press, 307–317.
[45] A.R. Sharafat and O.R. Marouzi. 2006. Recursive contraction algorithm: A novel and efficient graph traversal method for scanning all minimal cut sets. *Iranian Journal Of Science And Technology, Transaction B: Engineering* 30 (2006), 749–761.
[46] Oskar Skibski, Tomasz P. Michalak, and Talal Rahwan. 2018. Axiomatic Characterization of Game-Theoretic Centrality. *Journal of Artificial Intelligence Research* 62 (2018), 33–68.
[47] Oskar Skibski, Tomasz P. Michalak, Talal Rahwan, and Michael Wooldridge. 2014. Algorithms for the Shapley and Myerson values in graph-restricted games. In *Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IFAAMAS, 197–204.
[48] Oskar Skibski, Talal Rahwan, Tomasz P. Michalak, and Makoto Yokoo. 2016. Attachment Centrality: An Axiomatic Approach to Connectivity in Networks. In *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IFAAMAS, 168–176.
[49] Oskar Skibski and Makoto Yokoo. 2017. An Algorithm for the Myerson Value in Probabilistic Graphs with an Application to Weighted Voting. *IEEE Intelligent Systems* 32, 1 (2017), 32–39.
[50] Jadwiga Sosnowska and Oskar Skibski. 2017. Attachment Centrality for Weighted Graphs. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI, 416–422.
[51] Francesca Spezzano, VS. Subrahmanian, and Aaron Mannes. 2014. Reshaping terrorist networks. *Commun. ACM* 57, 8 (2014), 60–69.
[52] Katia Sycara, Massimo Paolucci, Martin Van Velsen, and Joseph Andrew Giampapa. 2003. The RETSINA MAS infrastructure. *Autonomous Agents and Multi-Agent Systems* 7, 1/2 (2003), 29–48.
[53] Thomas Voice, Sarvapali D. Ramchurn, and Nicholas R. Jennings. 2012. On coalition formation with sparse synergies. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IFAAMAS, 223–230.
[54] Duncan J. Watts and Steven H. Strogatz. 1998. Collective dynamics of small-world networks. *Nature* 393, 6684 (1998), 440–442.
[55] Michael Wooldridge. 2009. *An introduction to multiagent systems*. John Wiley & Sons.
[56] Wayne W Zachary. 1977. An information flow model for conflict and fission in small groups. *Journal of anthropological research* 33, 4 (1977), 452–473.

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*.

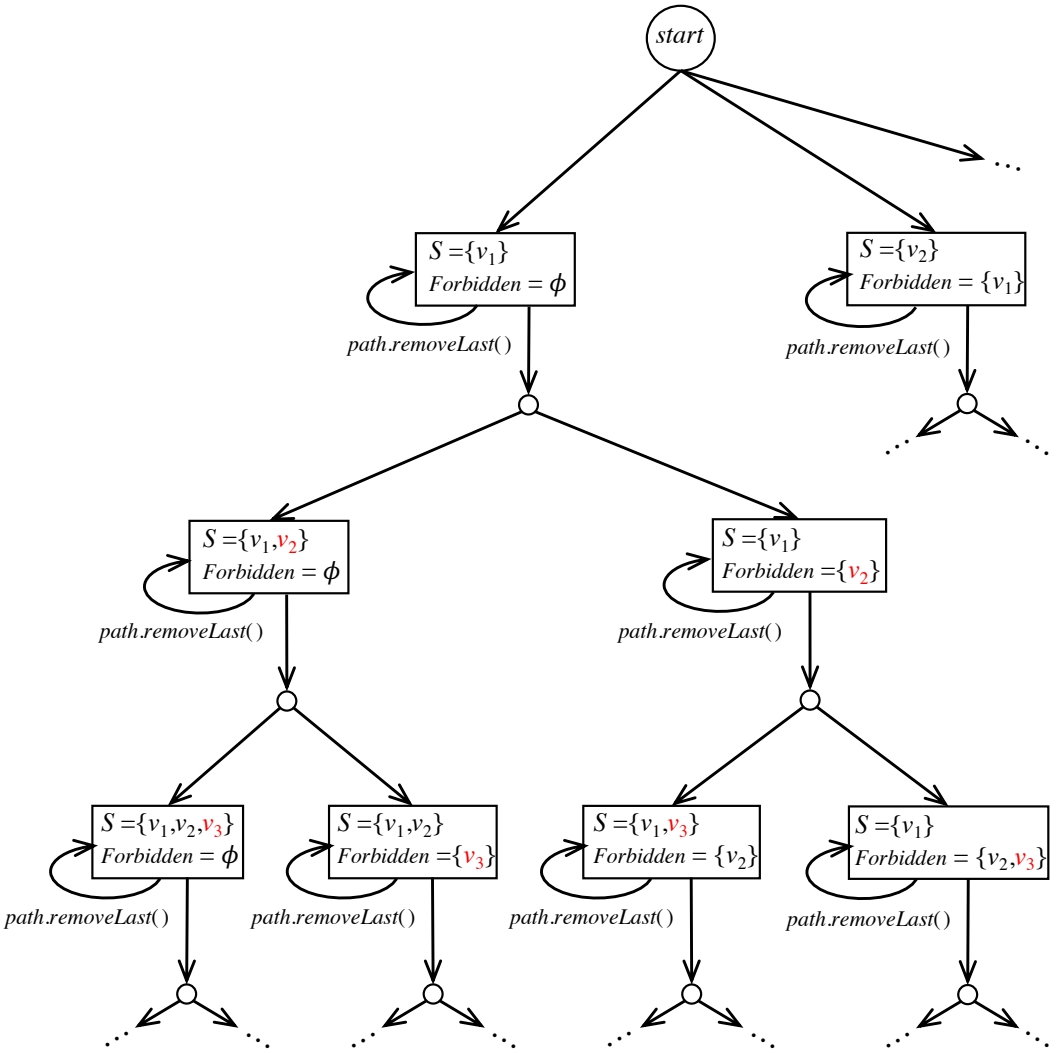


Fig. 4. A graphical overview of the workings of DFSE, which illustrates how the state of (S, *Forbidden*) is changed during the execution of the recursive function *ExpandSubgraph*. Here, every rectangle represents a certain state of (S, *Forbidden*). For every such state, the process can be divided 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.

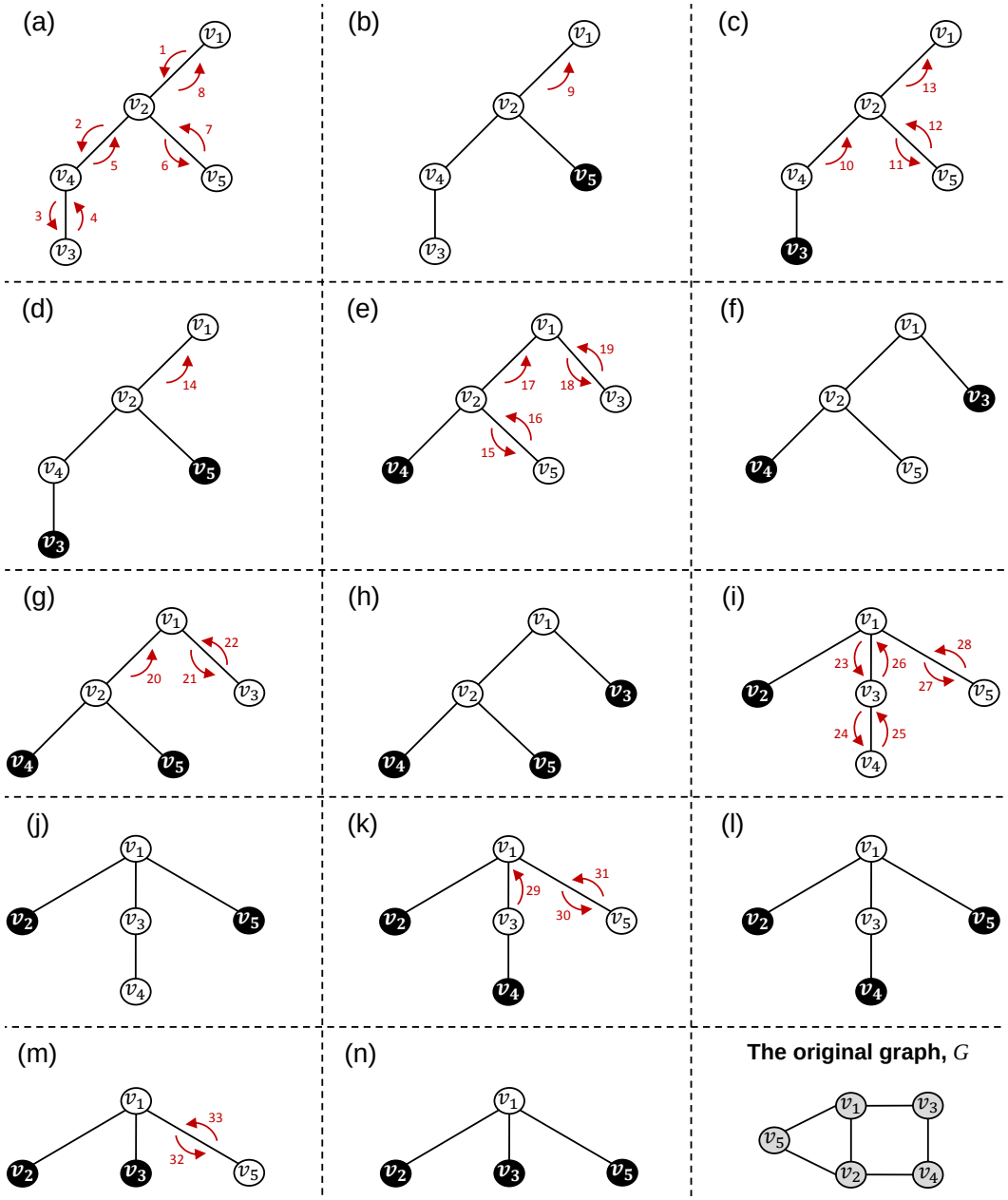


Fig. 5. A sample graph, G, along with an illustration of how DFSE generates the subgraphs that contain v_1 (the order of the subgraphs follows the order in which DFSE generates those subgraphs). In each subfigure, the white nodes are those in the subgraph, S, whereas the black nodes are those in *Forbidden*. 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).

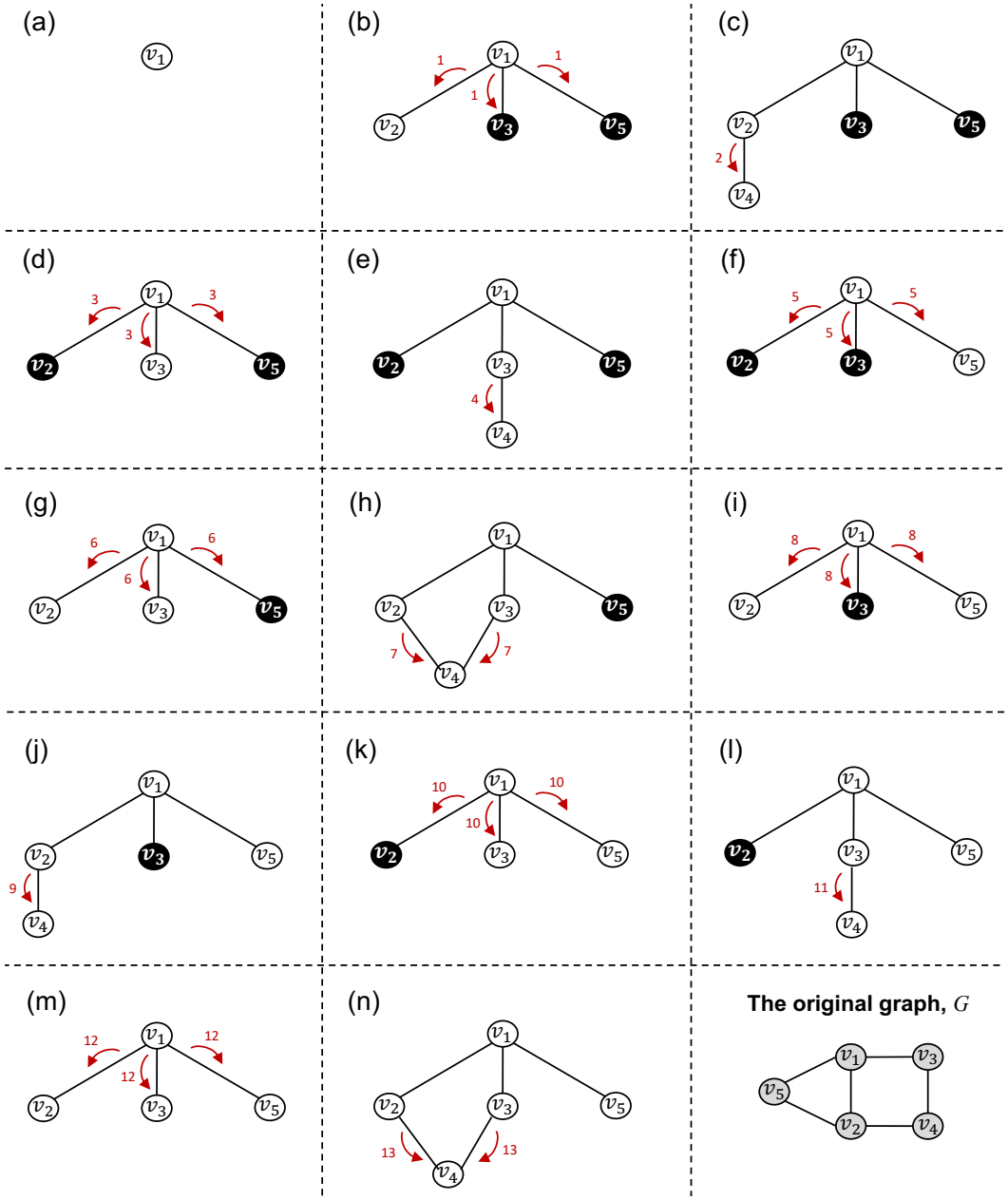


Fig. 6. A sample graph, G, along with an illustration of how BFSE generates the subgraphs that contain v_1 (the order of the subgraphs follows the order in which BFSE generates those subgraphs). In each subfigure, the white nodes are those in the subgraph, $Old \cup New$, whereas the black nodes are 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

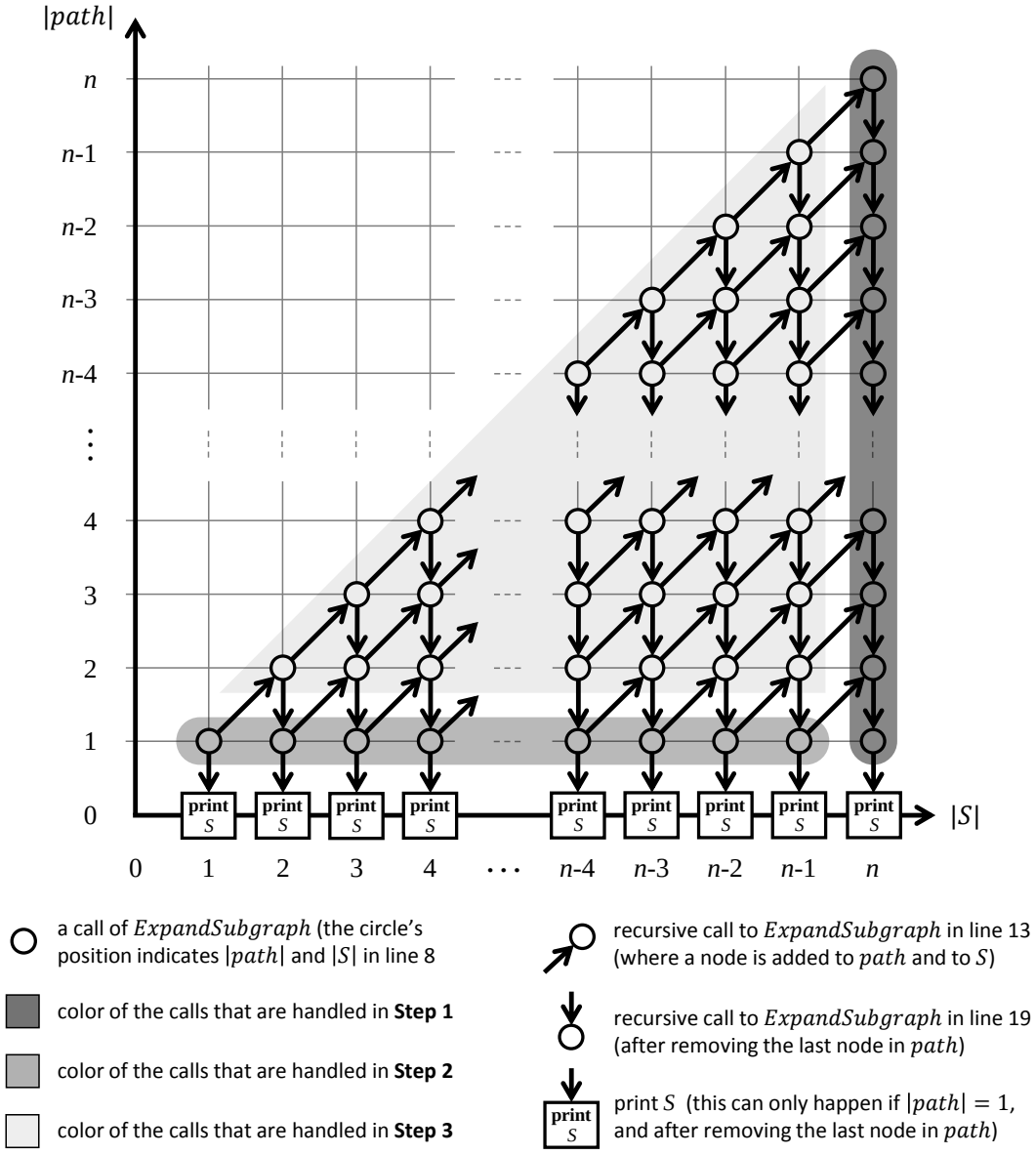


Fig. 7. In this plot, a circle located at (x, y) represents a call to *ExpandSubgraph* where $|S| = x$ and $|path| = y$ in line 8. Here, the recursive call in line 13 (which can only be made if $|S| < n$ in line 8) is represented by a diagonal arrow pointing towards the circle located at $(|S| + 1, |path| + 1)$; 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.

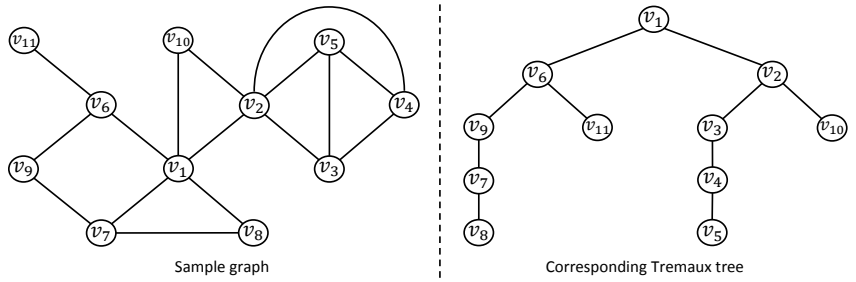


Fig. 8. A sample graph and its corresponding Trémaux tree. As is done by the DFSE algorithm, the nodes in the graph are sorted descendingly based on their degree, and are re-indexed accordingly (ties are broken uniformly at random).

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:

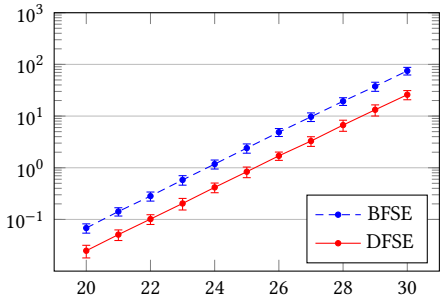
$$\begin{aligned} &\text{(i)} \quad f_G(S) = \frac{|E(S)|}{\sum_{\{v_i,v_j\}\in E(S)} \omega_{ij}}, \qquad \text{(ii)} \quad f_G(S) = \sum_{v_i\in S} \omega_i, \\ &\text{(iii)} \quad f_G(S) = \max_{\{v_i,v_j\}\in E(S)} \omega_{ij}, \quad \text{(iv)} \quad f_G(S) = \left(\max_{\{v_i,v_j\}\in E(S)} \omega_{ij}\right)\left(\sum_{v_i\in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

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].

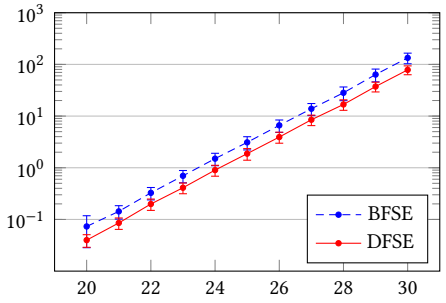
APPENDIX: EXTENDED EXPERIMENTAL ANALYSIS

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.

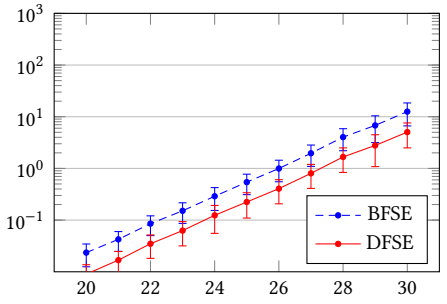
Received xxx 2018; revised xxx 2019; accepted xxx 2019



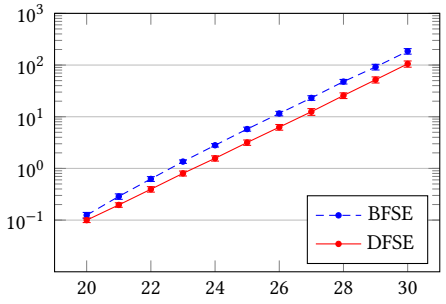
(a) Scale-free graphs (number of edges added with each node = 4).



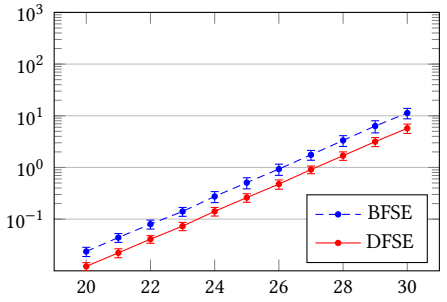
(b) Scale-free graphs (number of edges added with each node = 10).



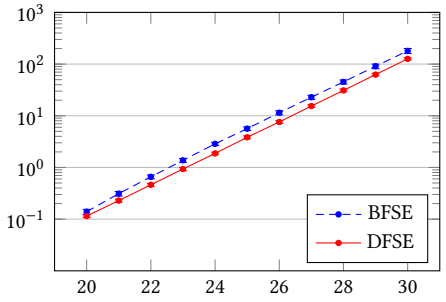
(c) Erdős-Rényi graphs (expected average degree = 4).



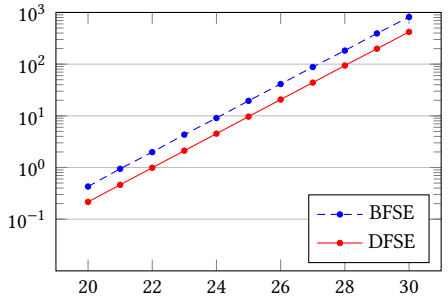
(d) Erdős-Rényi graphs (expected average degree = 10).



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

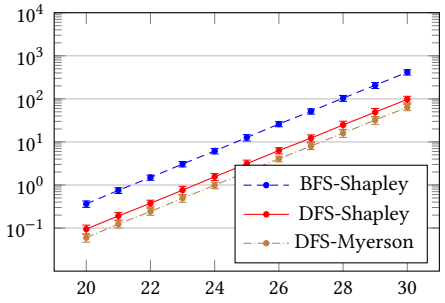


(f) Small-world graphs (expected degree = 10; rewiring prob. = 0.25).

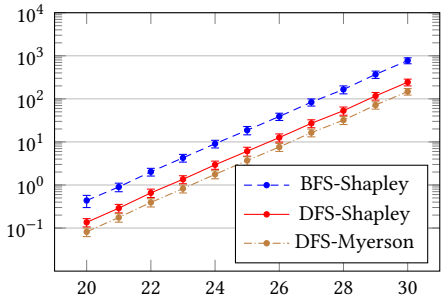


(g) Complete graphs.

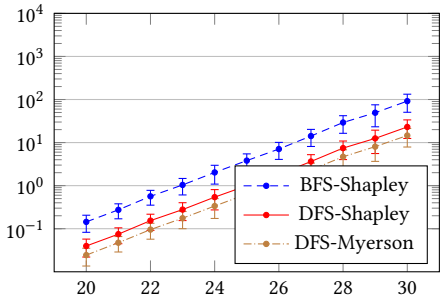
Fig. 9. Given different numbers of players (the x-axis), the figure depicts the time required to enumerate all connected induced subgraphs (the y-axis). For each random-network setting, the results are averaged over 100 different networks. DFSE was introduced in Algorithm 1. BFSE was introduced by [34]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



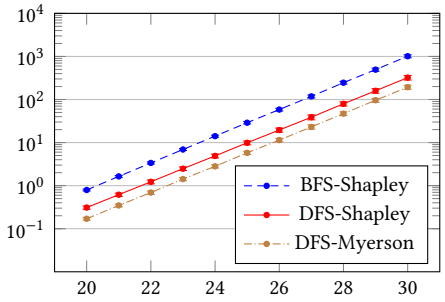
(a) Scale-free graphs (number of edges added with each node = 4).



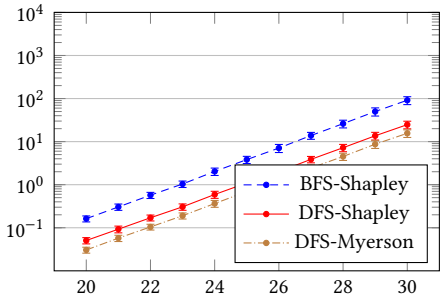
(b) Scale-free graphs (number of edges added with each node = 10).



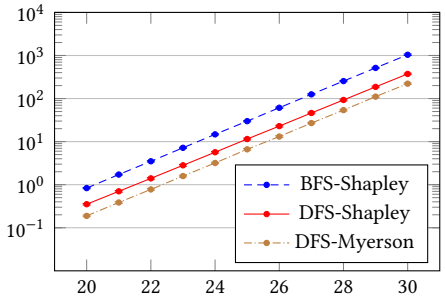
(c) Erdős-Rényi graphs (expected average degree = 4).



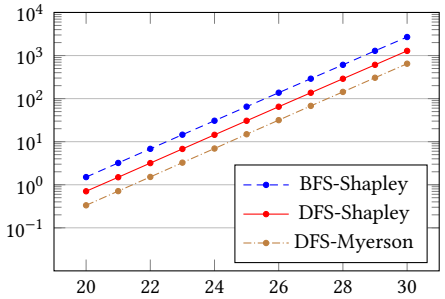
(d) Erdős-Rényi graphs (expected average degree = 10).



(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].