A dynamic programming algorithm for the maximum s-club problem on trees

José Alberto Fernández-Zepeda[1], Alejandro Flores-Lamas[2], Matthew Hague[3], and Joel Antonio Trejo-Sánchez[4]

[1] Department of Computer Science, Center for Scientific Research and Higher Education of Ensenada (CICESE)
joel.trejo@cimat.mx
[2] Department of Computer Science, Royal Holloway, University of London
Alejandro.Flores-Lamas@rhul.ac.uk
[3] Department of Computer Science, Royal Holloway, University of London
Matthew.Hague@rhul.ac.uk
[4] SECIHTI-Centro de Investigación en Matemáticas (CIMAT)
fernan@cicese.mx

Abstract

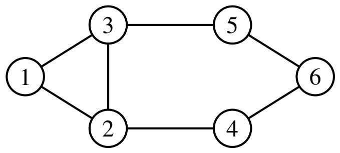
Computing cliques in an undirected graph $G = (V_G, E_G)$ is a fundamental problem in social network analysis. However, in some cases, the strict definition of a clique (a subset of vertices pairwise adjacent in G) often limits its applicability in real-world settings. To address this issue, we study the s-club: a clique relaxation that induces a subgraph of diameter at most s. Note that a clique is simply a 1-club. Computing a maximum s-club is a computationally challenging problem, as it is NP-hard for any positive integer s in arbitrary graphs. Thus, this paper presents a simple dynamic programming algorithm that efficiently computes a maximum s-club on an n-vertex tree in $O(s \cdot n)$ time. This algorithm outperforms existing algorithms for trees in theory and practice. This approach is a stepping stone towards computing maximum s-clubs on tree-like graphs.

1 Introduction

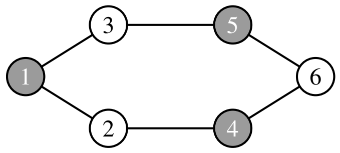
Let $G = (V_G, E_G)$ be an undirected graph of order $|V_G| = n$ and size $|E_G| = m$. A clique $C \subseteq V_G$ is a subset of vertices pairwise adjacent in G. Finding cliques is a crucial problem in graph theory with various applications in biological networks, social sciences, community detection, and combinatorial optimisation. However, in some cases, the strict definition of the clique can be too rigid to model many real-world phenomena. Researchers have proposed various clique relaxations to address this issue, one of which is the s-club.

For a fixed integer $s > 0$, a subset $S \subseteq V_G$ is an *s-club* if the diameter of the subgraph induced by S is at most s. The s-club is a diameter-based relaxation of the clique: a clique induces a subgraph of diameter exactly 1, whereas an s-club allows a diameter of at most s. Note that a clique is also a 1-club. Related clique relaxations include s-cliques, where the distance between any two vertices in S is at most s in the original graph G, and k-plexes, which relax vertex degree constraints [41]. These clique relaxations are relevant because they identify previously omitted relations due to the restrictiveness of the clique.

A set is called *maximum* with respect to a given property if it satisfies that property and has the largest possible cardinality among all such sets in the graph. This definition contrasts with a *maximal* set, which is defined as a set that is not strictly contained in any larger set satisfying the same property. These notions naturally apply to cliques, s-cliques, and s-clubs. Thus, the maximum s-club (clique, s-clique) problem consists of computing an s-club (clique,



(a) Sets $A = \{1, 2, 3, 4, 5\}$ and $B = \{2, 3, 4, 5, 6\}$ are the maximum 2-cliques. Set B is the maximum 2-club. Sets $C = \{1, 2, 3, 4\}$ and $D = \{1, 2, 3, 5\}$ are maximal 2-clubs. Example provided in [34].



(b) Clear and grey vertices form disconnected maximum 2-cliques [2].

Figure 1: Examples of 2-cliques and 2-clubs.

s-clique) of maximum cardinality in G. The decision version of this problem, denoted by the s-Club problem, consists of deciding if there exists an s-club of cardinality at least k in G.

Figure 1a shows examples of 2-cliques and 2-clubs in a graph. An s-club is more restrictive than an s-clique since the graph induced by an s-clique is not necessarily connected and does not necessarily have a diameter of size s. Figure 1b shows a graph containing disconnected maximum 2-cliques. Furthermore, unlike cliques and s-cliques, s-clubs are not hereditary, meaning each subset of an s-club is not necessarily an s-club [38]. For instance, in Figure 1a, the set $E = \{1, 2, 5\} \subset D$ is not a 2-club.

This paper presents DPsC ('dynamic programming s-club'), a simple dynamic programming algorithm that solves the maximum s-club problem in $O(s \cdot n)$ time when G is an arbitrary tree. We show that DPsC improves upon previous algorithms both theoretically and experimentally. Please note that any s-clique is also an s-club on tree graphs, as tree structures ensure that the shortest path between any two vertices is unique and remains within the induced subgraph.

Computing s-clubs have diverse applications, including community detection [19, 30], biological networks [40], graph covering [14], graph partitioning [54], clustering biological networks [6], crime detection/prevention [5], [44], and bike-sharing systems [43]. While applications employ s-cliques, they could benefit from the internal connectivity of s-clubs. For instance, s-club-based clustering has potential in protein sequence analysis [33] and genome mapping [21].

Since the maximum s-club problem is known to be NP-hard on an arbitrary graph [10], it is interesting to look for restricted classes of graphs where the problem becomes tractable. This paper deals with trees, which are fundamental in constructing efficient data structures. The recursive nature of trees makes them well-suited to problems that require decomposing complex tasks into smaller and more manageable subproblems, a quality also desirable in other graphs. In addition, tasks such as sorting, searching, data storing, and data compression use trees extensively. Moreover, tree-based algorithms can often serve as subroutines for solving problems on larger graph classes, such as unicyclic graphs and cacti [17].

Trees also find applications in diverse fields, from theoretical computer science and machine learning to representing company hierarchies and the structure of biological organisms. Trees also play a pivotal role in computational biology and phylogenetics. Phylogenetic trees model evolutionary relationships between species or genes and are fundamental in conservation biology, disease tracking, forensics, and protein function prediction [36, 1]. In [18], the author discusses community detection in hierarchical structures like trees. In this context, s-clubs can support clustering methods that group similar organisms based on evolutionary distances

[16, 42]. Examples of such clustering methods include hierarchical, k-means, k-medoid, and density approaches [31].

Figure 2 shows an example where clades (groups of organisms that descend from a common ancestor) in a phylogenetic tree naturally form s-clubs. The Dinosauria clade, shown within the dotted rectangle, is a 4-club since the largest distance between the 'birds' node and any of the three 'non-avian dinosaurs' nodes is 4-edge long. Similarly, the Archosauria clade (enclosed by dashed lines) is a 5-club since the distance between the 'birds' node and the 'crocodiles' node is 5-edge long.

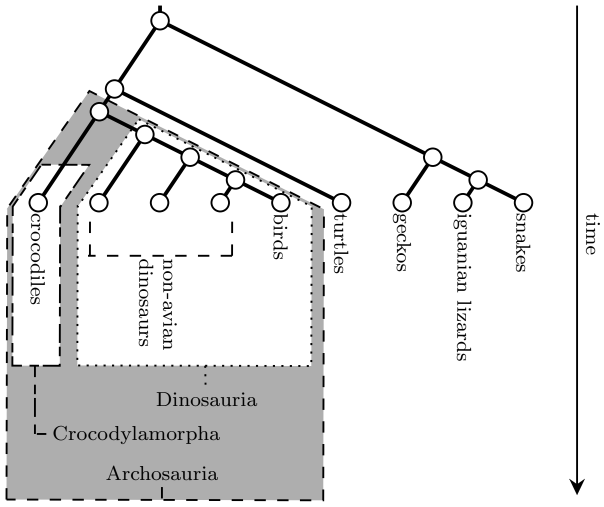


Figure 2: Phylogenetic tree adapted from [28]. The Dinosauria clade is a 4-club. The Archosauria clade is a 5-club.

The remaining sections of this paper are the following: Section 2 covers basic graph concepts. Section 3 reviews previous work on s-clubs. Section 4 describes DPsC and its terminology. Section 5 presents the correctness and analysis of our algorithm. Section 6 outlines implementation and experimental results. Finally, Section 7 provides some concluding remarks and future work.

2 Preliminaries

A *graph* G is a tuple (V_G, E_G), where V_G is a finite set of vertices and E_G is a binary relation of those vertices. The graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V_G$, and $E' \subseteq E_G$. In particular, if E' contains all the edges $(u, v) \in E$ such that $u, v \in V'$, then G' is an *induced subgraph* of G and is denoted by $G[V']$.

Let $[\chi] = \{0, 1, \ldots, \chi\}$ be the closed interval from 0 to χ. A *path* $v_0 P v_\chi$ in G is a sequence of distinct vertices $\langle v_0, v_1, \ldots, v_{\chi-1}, v_\chi \rangle$, where $v_i \in V_G$ for all $i \in [\chi]$, and there exists an edge in E_G for each pair of consecutive vertices in the path. The length of a path is the number of edges it contains. The *shortest path* between two vertices u and v is a path that has the minimum number of edges among all the paths connecting u and v in G. The distance between two vertices u and v in G, denoted by $dist_G(u, v)$, is the number of edges in a shortest path connecting them. The *diameter* of G, denoted by $diam(G)$, is the greatest distance between any two of its vertices.

A *tree* $T = (V_T, E_T)$ is a connected acyclic graph. A *rooted tree* is a tree with a special vertex called the *root*. Let z be the root of T and (u, v) the last edge on the path zPv, then u is the *parent* of v and v is a *child* of u. A vertex v with no children is a *leaf*; otherwise, v is an internal vertex. Let T be a rooted tree with root z, $v \in V_T$, and $v \neq z$, then the tree rooted at v consists of the subtree of T induced by v and all its descendant vertices. A *postorder traversal* is a method that visits the vertices of a tree rooted at z in the following order: first, recursively traverse from left to right all the subtrees rooted at the children of z and then visit z.

3 Related work

In this section, we first recall some key algorithmic results for the maximum clique problem and then review known results related to the maximum s-club problem.

Although solving the maximum clique problem in a graph is NP-hard [22], there exist several exact exponential-time algorithms to solve this problem, including the well-known algorithm by [11], and its subsequent optimisations [37]. Other works have proposed approximation algorithms for the maximum clique problem [15], yet it remains NP-hard to approximate within a factor of $n^{1-\epsilon}$ for any $\epsilon > 0$ [25]. In addition, no fixed-parameter tractable (FPT) approximation algorithm is currently known [13]. We refer the reader to the survey by [53] for a comprehensive overview of exact, heuristic, and metaheuristic methods, and to the preprint by [32] for a summary of classical algorithms as well as recent developments involving graph neural networks and quantum approaches.

The maximum s-club problem is also NP-hard in general graphs [10]. They also provide an exact branch-and-bound algorithm to solve the maximum s-club problem. Even more, testing the maximality of an s-club is also NP-hard [38]. Subsequent studies have proposed algorithms for solving the maximum s-club problem [9, 10, 48], and [52] generalized the concept of s-club for edge-weighted graphs. [55] developed exact algorithms to solve the maximum biconnected and fragile 2-club problems.

Regarding approximation algorithms for the maximum s-club problem, [4] proved that computing a maximum s-club is not approximable within a factor of $O(n^{1/2-\epsilon})$, for any $\epsilon > 0$. They also proposed an $n^{1/2}$-approximation algorithm for this problem.

Some relevant contributions dealing with FPT algorithms for the s-Club problem are [47], [24], and [8]. Later, [23] proposed an $O(n)^{f(m)}$ XP algorithm for the 2-Club problem, where m is a constant representing the h-index of the input graph, a parameter related to the vertex degree.

Researchers have extensively explored integer programming formulations for the maximum s-club problem [6]. [3] designed a compact ILP model specifically for the 3-club case, and [35] developed a decomposition-based branch-and-cut approach capable of solving larger instances. [12] and [39] analyzed the polyhedral properties of the 2-club problem, while [51] introduced ILP formulations tailored to sparse graphs for $s = 2$. For a comprehensive overview of s-club research, we refer the reader to the survey by [7].

Among the most influential general-purpose models, [50] introduced a recursive ILP formulation that reduces the size of earlier models from $O(n^{s+1})$ to $O(s \cdot n^2)$ variables and constraints. They also introduced the robustness-based (s, R)-club variant, which guarantees at least R different paths between each pair of vertices in the s-club. This model is inapplicable to our work due to unique paths between tree vertices.

More recently, [45] proposed the cut-like integer programming formulation. It uses only n binary variables and a single class of restrictions. This formulation outperforms previous approaches regarding scalability and solver performance for the maximum s-club problem.

The work most closely related to ours is by [46], who presented an exact algorithm for computing a maximum s-club on an arbitrary n-vertex tree in $O(s^2 \cdot n)$ time. The author also presented similar algorithms for other graph classes. Section 4.4 presents a detailed comparison between this algorithm and our approach, DPsC.

4 The DPsC algorithm

The DPsC(T, v_{root}, s) algorithm aims to compute a maximum s-club, denoted by $\hat{S}$, within an arbitrary tree T. The algorithm uses the ROOT function with parameters T and v_{root} to optimally root T at vertex v_{root}. DPsC combines the functionality of two sub-algorithms, UPWARD-SWEEP and DOWNWARD-SWEEP, to efficiently compute and retrieve $\hat{S}$. Pseudocode 1 shows the DPsC algorithm.

- UPWARD-SWEEP: DPsC begins by invoking UPWARD-SWEEP, which processes T rooted at v_{root} in a postorder traversal (see Section 4.1). This process involves computing the cardinalities of the best subproblem solutions and storing them in tables, one for each T vertex. The objective identifying which vertex table and row store $|\hat{S}|$. We refer to this vertex and row as $\hat{v}$ and r, respectively.
- DOWNWARD-SWEEP: With $\hat{v}$ and r identified, DPsC then calls DOWNWARD-SWEEP, which traces back through the pointers stored during the execution of UPWARD-SWEEP. This recursive procedure reconstructs the set of vertices in $\hat{S}$.
- Output: The algorithm returns $\hat{S}$. In particular, $T[\hat{S}]$ is the subtree of T rooted at $\hat{v}$ whose height is r.

Pseudocode 1: DPsC(T, v_{root}, s)

```
Input  : An arbitrary tree $T = (V_T, E_T)$, an arbitrary vertex $v_{root} \in V_T$, and an integer $s \geq 2$.
Output: A set $\hat{S} \subseteq V_T$ that represents the vertices of a maximum $s$-club of $T$.
1 begin
2     $T \leftarrow$ ROOT$(T, v_{root})$          ▷ Now, each vertex $v \in V_T$ knows its
       parent and children
3     $(\hat{v}, r) \leftarrow$UPWARD-SWEEP$(T, s)$
4     $\hat{S} \leftarrow \{\emptyset\}$
5     $\hat{S} \leftarrow$ DOWNWARD-SWEEP $\left(T, \hat{v}, r, \hat{S}\right)$
6     return $\hat{S}$
7 end
```

4.1 Terminology and notation

We denote vertices by lower-case letters and edges by an unordered pair of vertices. Let $T = (V_T, E_T)$ be the input graph to DPsC, an arbitrary undirected tree. Since DPsC rooted tree T at vertex v_{root}, from here onwards assume that each vertex $v \in V_T$ knows its parent $p(v)$ and children $C_v = \{u_0, u_1, \ldots, u_i, \ldots, u_{|C_v|-1}\}$. Let $T_v \subseteq T$ be the subtree rooted at vertex v, and $T_v^{[r]} \subseteq T_v$ be the subtree containing v and its descendants up to a distance $r \leq s$. Let $\hat{S}_v$ ($\hat{S}_v^{[r]}$) be any set of vertices with the largest cardinality such that $\hat{S}_v$ ($\hat{S}_v^{[r]}$) is a subset of the vertices of T_v ($\hat{T}_v^{[r]}$) and the $dist(u, w) \leq s$ for $u, w \in \hat{S}_v$ ($\hat{S}_v^{[r]}$). Finally, $\hat{T}_v = T[\hat{S}_v]$ ($\hat{T}_v^{[r]} = T[\hat{S}_v^{[r]}]$).

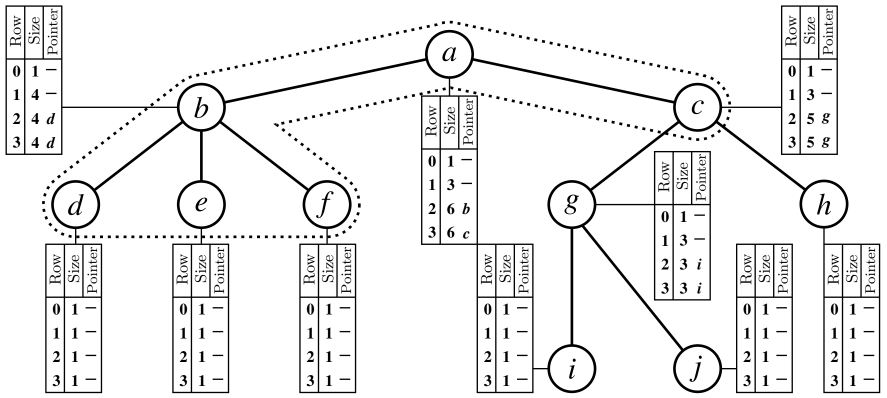

a
b
c
d
e
f
g
h
i
j
Row Size Pointer
b: 0 1 –; 1 4 –; 2 4 d; 3 4 d
a: 0 1 –; 1 3 –; 2 6 b; 3 6 c
c: 0 1 –; 1 3 –; 2 5 g; 3 5 g
g: 0 1 –; 1 3 –; 2 3 i; 3 3 i
d, e, f, i, j, h: 0 1 –; 1 1 –; 2 1 –; 3 1 –

Figure 3: An execution example of DPsC on a tree T rooted at vertex a. The subtree inside the dotted lines is a maximum 3-club.

DPsC uses dynamic programming and assigns a table to each vertex v, denoted as $v.table$. Then, $v.table[r].Data$ refers to the entry in row r, where $r \in [s]$ and $Data \in \{Size, Pointer\}$. Each entry $v.table[r].Size$ stores $|\hat{S}_v^{[r]}|$, and $v.table[r].Pointer$ keeps a single vertex that helps to identify the vertices belonging to $\hat{S}$ during DOWNWARD-SWEEP. It is essential to highlight that a maximum s-club in T is not necessarily unique. For our purposes, it is enough to output just one of them. See Remark 4.1 and Example 4.2.

Remark 4.1. UPWARD-SWEEP *follows a postorder traversal on T and returns the pair $(\hat{v}, r)$. Vertex $\hat{v}$ has the largest postorder number, and r is the row with the smallest number in $\hat{v}.table$, such that the value of $|\hat{S}_{\hat{v}}^{[r]}|$, stored in $\hat{v}.table[r].Size$, has the largest cardinality in T; i.e., $|\hat{S}_{\hat{v}}^{[r]}| = |\hat{S}|$.*

Example 4.2. ***Vertex table.*** *Figure 3 shows the table values of vertices $\{a, b, \dots, j\}$ in tree T after computing a maximum 3-club. The value 6 in entry $a.table[2].Size$ (the maximum in all entries) indicates that a subtree $\hat{T}_a^{[2]}$ exists such that $|\hat{S}_a^{[2]}| = |\hat{S}| = 6$ and $diam(\hat{T}_a^{[2]}) \leq 3$. The subtree $\hat{T}_a^{[2]}$, inside the dotted lines in Figure 3, is a maximum 3-club of T. Additionally, the entry $a.table[3].Size$ shows that the subtree $\hat{T}_a^{[3]}$ is also a maximum 3-club of T; however, by Remark 4.1,* UPWARD-SWEEP *returns the pair $(a, 2)$, indicating that the maximum-cardinality solution is 6.*

4.2 Upward-sweep

The UPWARD-SWEEP algorithm (see Pseudocode 2) aims to compute the cardinality of a maximum s-club in T. The output of the algorithm is the pair $(\hat{v}, r)$; i.e., the algorithm stores $|\hat{S}|$ in the table row r associated with vertex $\hat{v}$. Recall from Subsection 4.1 that each vertex has a table with rows $r \in [s]$ for integers r and s. The algorithm operates through a postorder traversal of the tree, processing each vertex v after finishing the processing of all children of v.

The algorithm begins by filling in the table for leaf vertices, where the cardinality of the maximum s-club is trivially one because a leaf vertex has no children. (See Equation 1 and Line 4 of Pseudocode 2.)

$$v.table[r].Size = 1, \quad \text{if } v \text{ is a leaf vertex, for } r \in [s]. \tag{1}$$

The algorithm computes the maximum s-club for each non-leaf vertex v. Such a computation considers four cases based on the r-value, adhering it to Equations 2 to 8.

- **Case 0**: Similar to the leaves, the size of the maximum s-club for row $r = 0$ is one since it only includes vertex v. (See Equation 2 and Line 7 of Pseudocode 2.)

$$v.table[0].Size = 1, \quad \text{if } v \text{ is an internal vertex.} \tag{2}$$

- **Case 1**: For rows $r = 1$ up to $\left\lfloor \frac{s}{2} \right\rfloor$, the algorithm sums the optimal solutions computed by its children stored at row $r-1$ and adds 1 for the current vertex, i.e., v. (See Equation 3 and Line 8 of Pseudocode 2.)

$$v.table[r].Size = 1 + \sum_{\forall u \in C_v} u.table[r-1].Size, \text{ if } v \text{ is an internal vertex, for } 1 \le r \le \left\lfloor \frac{s}{2} \right\rfloor. \tag{3}$$

- **Case 2**: For rows greater than $\left\lfloor \frac{s}{2} \right\rfloor$ and less than $s-1$, the algorithm calculates the contribution from the optimal solution stored at row $r-1$ in a child vertex u, plus the sum of the optimal solutions of all brothers of u at row $s-r-1$. That is to say, the distance from any vertex in the s-club of u to any other optimal solution in $C_v \setminus u$ stored at row $s-r-1$ will be at most s edges long. The algorithm iterates through all child vertices to maximise this contribution and adds one for the current vertex, i.e., v. (See Equations 4-6 and Lines 10-19 of Pseudocode 2.)

$$\alpha = \sum_{\forall u \in C_v} u.table[s-r-1].Size, \text{ if } v \text{ is an internal vertex.} \tag{4}$$

$$v.table[r].Size = 1 + \max_{\forall u \in C_v} \left(\begin{array}{c} \alpha \\ - u.table[s-r-1].Size \\ + u.table[r-1].Size \end{array} \right), \quad \begin{array}{l} \text{if } v \text{ is an internal vertex,} \\ \text{for } \left\lfloor \frac{s}{2} \right\rfloor < r \le s-1. \end{array} \tag{5}$$

$$v.table[r].Pointer = u^*, \quad \text{if } v \text{ is an internal vertex, for } \left\lfloor \frac{s}{2} \right\rfloor < r \le s-1. \tag{6}$$

- **Case 3**: It considers the setting where an optimal solution for the vertex v (to store in the row $r = s$) is one (i.e., vertex v) plus the cardinality of the optimal solution stored at row $s-1$ of v's children. (See Equations 7-8 and Lines 21-29 of Pseudocode 2.)

$$v.table[s].Size = 1 + \max_{\forall u \in C_v} \left(u.table[s-1].Size \right), \quad \text{if } v \text{ is an internal vertex.} \tag{7}$$

$$v.table[s].Pointer = u^*, \quad \text{if } v \text{ is an internal vertex.} \tag{8}$$

Pseudocode 2: UPWARD-SWEEP(T, v_{root}, s)

```
Input  : A tree $T = (V_T, E_T)$ rooted at some vertex $v_{root} \in V_T$, and an integer $s \geq 2$.
Output: A pair $(\hat{v}, r)$, such that $\hat{v}.table[r].Size$ stores $|S_v^{[r]}| = |\hat{S}|$. See Remark 4.1.
begin
    foreach $v \in V_T$ in POSTORDER$(T, v_{root})$ do
        if $v$ is a leaf vertex then
            foreach $r \in [s]$ do $v.table[r].Size \leftarrow 1$
        else
            $u^* \leftarrow \emptyset$
                                                        ▷ Case 0.
            $v.table[0].Size \leftarrow 1$
                                                        ▷ Case 1.
            for $1 \leq r \leq \lfloor \frac{s}{2} \rfloor$ do $v.table[r].Size \leftarrow 1 + \sum_{\forall u \in C_v} u.table[r-1].Size$
                                                        ▷ Case 2.
            for $\lfloor \frac{s}{2} \rfloor < r \leq s-1$ do
                $\alpha \leftarrow \sum_{\forall u \in C_v} u.table[s-r-1].Size$
                $ans \leftarrow 0$
                foreach $u \in C_v$ do
                    if $\alpha - u.table[s-r-1].Size + u.table[r-1].Size > ans$ then
                        $u^* \leftarrow u$
                        $ans \leftarrow \alpha - u.table[s-r-1].Size + u.table[r-1].Size$
                    end
                end
                $v.table[r].Size \leftarrow 1 + ans$
                $v.table[r].Pointer \leftarrow u^*$
            end
                                                        ▷ Case 3.
            $ans \leftarrow 0$
            foreach $u \in C_v$ do
                if $u.table[s-1].Size > ans$ then
                    $u^* \leftarrow u$
                    $ans \leftarrow u.table[s-1].Size$
                end
            end
            $v.table[s].Size \leftarrow 1 + ans$
            $v.table[s].Pointer \leftarrow u^*$
                                                        ▷ Optimal solution check.
            for $1 \leq r \leq s$ do
                if $v.table[r].Size < v.table[r-1].Size$ then
                    $v.table[r].Size \leftarrow v.table[r-1].Size$
                    $v.table[r].Pointer \leftarrow v.table[r-1].Pointer$
                end
            end
        end
    end
    return $(\hat{v}, r) \leftarrow \max_{\forall v \in V_T} (v.table[r].Size)$, according to Remark 4.1
end
```

After processing each row, the algorithm checks to ensure that the cardinality of the optimal solution stored in row r is not decreasing as r increases. (See Equation 9, Lines 30-35 of Pseudocode 2, and Example 4.6.)

$$v.table[r+1].Data = v.table[r].Data \quad \text{if } v.table[r+1].Size < v.table[r].Size, \\ \text{if } v \text{ is an internal vertex, for } r \in [s-1] \wedge Data \in \{Size, Pointer\}. \tag{9}$$

In Cases 2 and 3, the single pointer that UPWARD-SWEEP stores in $v.table[r].Pointer$ is vertex u^*. DOWNWARD-SWEEP uses this pointer to identify the vertices of $\hat{S}$ explicitly. For Cases 0 and 1, UPWARD-SWEEP does not save any pointer in $v.table[r].Pointer$.

Finally, after processing the entire tree, the algorithm identifies the vertex $\hat{v}$ and the row r that yields the maximum s-club (or one of them if multiple exist) in T. Note that this algorithm does not explicitly provide the set $\hat{S}$. It is necessary to trace back through the pointers stored in the tables to obtain $\hat{S}$.

Remark 4.3. *Equation 10 can concisely replace Equations 4 and 5; however, computing the former requires $O\left(|C_v|^2\right)$ time, whereas the time complexity for Equations 4 and 5 is $O\left(|C_v|\right)$.*

$$v.table[r].Size = 1 + \max_{\forall u \in C_v}\left(\sum_{\forall w \in C_v \setminus u} w.table[s-r-1].Size + u.table[r-1].Size\right). \qquad (10)$$

Example 4.4. ***Filling in a leaf vertex's table.*** *Consider the tree from Figure 3. Note that for each leaf vertex $v \in \{d,\ e,\ f,\ h,\ i,\ j\}$,* UPWARD-SWEEP *fills in each entry $v.table[r].Size$ with 1, for all r, according to Line 4 in Pseudocode 2. Leaf vertices do not require pointers, as indicated by the hyphen symbol ($-$) in entry $v.table[r].Pointer$.*

Example 4.5. ***Filling in an internal vertex's table.*** *Consider Examples 4.2 and 4.4 and the tree of Figure 3.* UPWARD-SWEEP *sets $a.table[0].Size \leftarrow 1$ and $a.table[1].Size \leftarrow 3$ according to Lines 7 and 8. Note that there are no pointers stored in such entries. To fill in entry $a.table[2].Size$, the algorithm selects a child $u^* \in C_a$ to read $u^*.table[r-1].Size = u^*.table[1].Size$, and for every $u \in \{C_a \setminus u^*\}$, it reads $u.table[s-r-1].Size = u.table[0].Size$.*

In this example, $\alpha \leftarrow b.table[0].Size + c.table[0].Size = 2$, by Line 10. Then, the algorithm identifies u^*, according to Lines 12-17. When the algorithm tries vertices b and c, variables $ans \leftarrow 2-1+4 = 5$ and $ans \leftarrow 2-1+3 = 4$, respectively. Thus, UPWARD-SWEEP *chooses $u^* \leftarrow b$ and sets $a.table[2].Size \leftarrow 6$ and $a.table[2].Pointer \leftarrow b$ according to Lines 18-19. Finally,* UPWARD-SWEEP *fills in a's table for $r = 3$. The **for** loop of Lines 22-27 selects the best child $u \in C_a$ to read entry $u.table[r-1].Size = u.table[2].Size$. When the algorithm tries vertices b and c, variables $ans \leftarrow 4$ and $ans \leftarrow 5$, respectively. Thus $u^* \leftarrow c$ and* UPWARD-SWEEP *sets $a.table[3].Size \leftarrow 6$ and $a.table[3].Pointer \leftarrow c$, by Lines 28-29.*

Example 4.6. ***Solution check.*** *Consider Examples 4.2 and 4.5 and the internal vertex c. The* UPWARD-SWEEP *algorithm fills in $c.table[3].Size \leftarrow 4$ and $c.table[3].Pointer \leftarrow g$ according to Equations 7 and 8. Since $c.table[2].Size > c.table[3].Size$,* UPWARD-SWEEP *updates $c.table[3].Size$ and $c.table[3].Pointer$ to the values of $c.table[2].Size$ and $c.table[2].Pointer$, respectively, according to Equation 9.*

4.3 Downward-sweep

The DOWNWARD-SWEEP algorithm (see Pseudocode 3) is a recursive procedure designed to trace back through the pointers stored (and computed by UPWARD-SWEEP) in the table of each vertex to reconstruct the set of vertices that form a maximum s-club $\hat{S}$. The input for this algorithm is the tree T, the pair (v, r), and an initially empty set $\hat{S}$. The algorithm begins by adding the current vertex v to the set $\hat{S}$ (see Line 2 of Pseudocode 3) and then recursively processes the children of v based on whether v is a leaf, in which the recursion ends (see Line 3 of Pseudocode 3); otherwise, similarly to UPWARD-SWEEP, it handles four cases based on the value of r:

Pseudocode 3: Downward-sweep$\left(T, v, r, \hat{S}\right)$

```
Input  : A rooted tree $T = (V_T, E_T)$, a vertex $v \in V_T$, a row $r$ from $v$'s table, and the set $\hat{S}$ to store
         the vertices of the computed maximum $s$-club.
Output: A set $\hat{S} \subseteq V_T$ that represents the vertices of the computed maximum $s$-club of $T$.
begin
    $\hat{S} \leftarrow \hat{S} \cup v$
    if $v$ is a leaf vertex then return $\hat{S}$
                                                                ▷ Case 0.
    if $r = 0$ then return $\hat{S}$
                                                                ▷ Case 1.
    if $1 \leq r \leq \lfloor \frac{s}{2} \rfloor$ then
        foreach $u \in C_v$ do
            $\hat{S} \leftarrow$ Downward-sweep$\left(T, u, r-1, \hat{S}\right)$
        end
    end
                                                                ▷ Case 2.
    if $\lfloor \frac{s}{2} \rfloor < r \leq s-1$ then
        $C_v \leftarrow C_v \setminus v.table[r].Pointer$
        $u^* \leftarrow v.table[r].Pointer$
        $\hat{S} \leftarrow$ Downward-sweep$\left(T, u^*, r-1, \hat{S}\right)$
        foreach $u \in C_v$ do
            $\hat{S} \leftarrow$ Downward-sweep$\left(T, u, s-r-1, \hat{S}\right)$
        end
    end
                                                                ▷ Case 3.
    if $r = s$ then
        $u^* \leftarrow v.table[r].Pointer$
        $\hat{S} \leftarrow$ Downward-sweep$\left(T, u^*, r-1, \hat{S}\right)$
    end
end
```

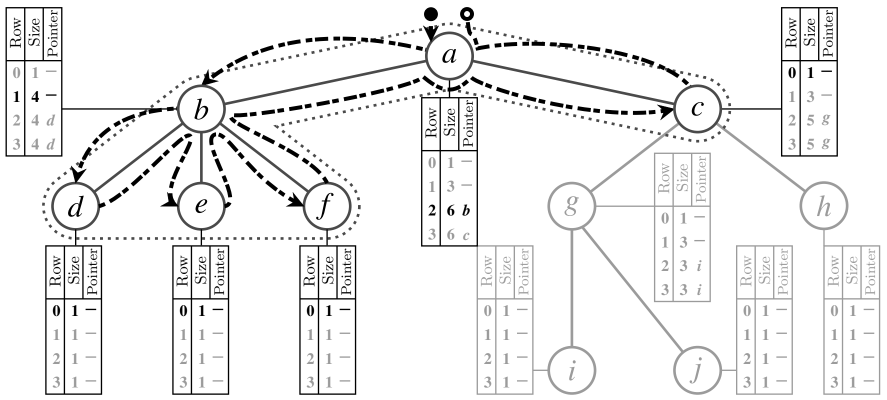
- Case 0: If $r = 0$, the recursion ends. (See Line 4 of Pseudocode 3.)
- Case 1: For $1 \leq r \leq \lfloor \frac{s}{2} \rfloor$, the algorithm processes all v's children recursively with $r - 1$. (See Lines 5-9 of Pseudocode 3.)
- Case 2: For $\lfloor \frac{s}{2} \rfloor < r \leq s - 1$, the algorithm first processes recursively with $r - 1$ the child vertex referred by the pointer stored in row r. Then, it processes recursively the remaining v's children with an $s - r - 1$. (See Lines 10-17 of Pseudocode 3.)
- Case 3: If $r = s$, the algorithm recursively processes the child vertex referred by the pointer (in row r) with $r - 1$. (See Lines 18-21 of Pseudocode 3.)

This algorithm ensures that $\hat{S}$ contains all the vertices of the maximum s-club by following the pointers identified during the Upward-sweep phase. Example 4.7 and Figure 4 illustrate how Downward-sweep works.

Example 4.7. Downward-sweep *is a preorder tree traversal in which the backtracking occurs when there are no more vertices to visit (i.e., when the visited vertex is a leaf or when row* $r = 0$*).*

Consider the tree T *rooted at vertex* a *of Figure 4. A preorder traversal visits the vertices in the order of* $\langle a, b, d, e, f, c, g, i, j, h \rangle$*; however,* Downward-sweep *only visits the* *vertices $\langle a,\ b,\ d,\ e,\ f,\ c\rangle$ because of its backtracking conditions. Using Remark 4.1,* Upward-sweep *on tree T returns the pair $(a, 2)$, as shown in Example 4.2. Therefore, the recursive approach of* Downward-sweep *starts with the input tree T, vertex a, row 2, and an initially empty set $\hat{S}$.* Downward-sweep *adds vertex a to $\hat{S}$ in Line 2. Since $r = 2$ is within the if statement's bounds, Line 10 evaluates to True and $a.table[r = 2].Pointer = b$ means that $u^* = b$ (Line 12). Thus,* Downward-sweep *visits the subtrees rooted at vertices b and c in preorder; however, r takes different values (1 and 0, respectively), i.e.,* Downward-sweep$(T, b, 1, \{a\})$ *and* Downward-sweep$(T, c, 0, \hat{S})$*, as shown in Lines 13 and 15.*

After visiting vertex b, $\hat{S}$ becomes $\{a, b\}$, and Downward-sweep *visits vertices $\langle d, e, f\rangle$ in preorder. $\hat{S}$ becomes $\{a,\ b,\ d,\ e,\ f\}$ at the end of such visits. The* Downward-sweep *call to visit vertex c is* Downward-sweep$(T,\ c,\ 0,\ \{a,\ b,\ d,\ e,\ f\})$*. During this preorder visit, $\hat{S}$ updates to $\{a,\ b,\ c,\ d,\ e,\ f\}$, and Line 4 returns this set since $r = 0$. The algorithm finishes in vertex a with $\hat{S} = \{a, b, c, d, e, f\}$ since all its children have been visited.*



Vertex	Row	Size	Pointer
b	0	1	–
b	**1**	**4**	**–**
b	2	4	d
b	3	4	d
a	0	1	–
a	1	3	–
a	**2**	**6**	***b***
a	3	6	c
c	**0**	**1**	**–**
c	1	3	–
c	2	5	g
c	3	5	g
g	0	1	–
g	1	3	–
g	2	3	i
g	3	3	i
d, e, f	**0**	**1**	**–**
d, e, f	1	1	–
d, e, f	2	1	–
d, e, f	3	1	–
h, i, j	0	1	–
h, i, j	1	1	–
h, i, j	2	1	–
h, i, j	3	1	–

Figure 4: Downward-sweep traversal on the tree of Figure 3 (after the execution of Upward-sweep). Dashed arrows' tips indicate recursive calls, and the bold and hollowed circles mark the initial and final states of the tree traversal, respectively. In this figure, Downward-sweep only uses the information in bold.

4.4 A comparison between the Schäfer's algorithm and DPsC

Schäfer's algorithm and DPsC efficiently compute the maximum s-club in an arbitrary tree, see Section 3. The two algorithms have similarities and differences in their approaches to solving this problem.

- **Similarities:** Both algorithms accept an integer s and a tree rooted at a unique vertex v_{root} as input. Each vertex has a table with $(s + 1)$ rows, and the algorithms perform a bottom-up traversal, starting with each vertex v to compute the cardinality $|\hat{S}|$ of vertices that form a maximum s-club. They also provide information to identify such vertices in a subsequent phase.
- **Differences:** Schäfer's algorithm builds tables similar to those in dynamic programming, where the entries represent the number of descendant vertices at a distance d from v. Such information does not necessarily represent the number of vertices in a specific s-club. In contrast, DPsC builds tables that store cumulative sums $\left(\text{representing } \hat{T}_v^{[d]}\right)$.

Schäfer's algorithm starts by computing the set of vertices at most at a distance s from the "lowest tree leaf" v. This computation involves adding data from $O(s)$ table entries and $O(s)$ distinct vertices. The resulting sum requires $O(s^2)$ operations. It represents the cardinality of the maximum s-club among all s-clubs in which v appears. Then, the algorithm removes v and repeats this process for the rest of the tree, keeping only the optimal solution.

DPsC, on the other hand, uses a dynamic programming strategy. For vertex v, it computes the cardinality of a maximum s-club, including v and some of its descendants, using the optimal solutions generated by the children of v. This computation involves only $O(s)$ operations per each vertex v, on average. DPsC follows a postorder traversal on T and repeats the above computations for each vertex to determine the cardinality of the maximum s-club.

Regarding efficiency, Schäfer's algorithm has an $O(s)$ overhead compared to DPsC.

5 Correctness and analysis

This section presents the correctness and analysis of DPsC. First, we prove its correctness by induction by showing that our dynamic programming algorithm correctly generates $O(s \cdot n)$ candidate solutions for $\hat{S}$. The algorithm stores the cardinality of each candidate solution (and, at most, a pointer per solution) in an entry of some of the n tables in T. Then, we use Equations 1-9, from Subsection 4.2, for its complexity analysis.

For this section, consider as input a tree $T = (V_T, E_T)$ as defined in Subsection 4.1. Let the height of $\hat{T}_v$ be the length of the longest path vPl, where l is a leaf vertex in $V_{\hat{T}_v}$. Also, recall that the $v.table[r].Size$ entry stores $|\hat{S}_v^{[r]}|$.

5.1 Correctness of DPsC

Next, we prove that the DPsC algorithm is correct and computes a maximum s-club, $\hat{S}$, at the end of its execution.

Lemma 5.1. Upward-sweep *correctly computes* $\hat{S}_v^{[r]}$ *for each vertex* v *in* T *and* $r \in [s]$*; i.e., it guarantees that* $|\hat{S}_v^{[r]}|$ *is maximum and* $diam(\hat{T}_v^{[r]}) \leq s$.

Proof. The proof is by induction on the postorder number of each vertex v in T.

Base case: Assume that the postorder number of vertex v is 1. Then, v is the leftmost leaf of T; therefore, $\hat{S}_v^{[r]}$ consists only of v, for any r. Equation 1 correctly sets $|\hat{S}_v^{[r]}| = 1$ in each entry of $v.table[r].Size$. Note that $\hat{S}_v^{[r]}$ is maximum and $diam(\hat{T}_v^{[r]}) = 0 < s$, for all $r \in [s]$.

Induction hypothesis: Assume that the Upward-sweep algorithm correctly computes the tables of each vertex v with postorder number less than or equal to p. As a result, for each v in this interval, $|\hat{S}_v^{[r]}|$ is maximum and $diam(\hat{T}_v^{[r]}) \leq s$, for all $r \in [s]$.

Inductive step: Assume that vertex v with postorder number $p+1$ is an internal vertex; otherwise, handle it as in the base case. We now prove that the Upward-sweep algorithm correctly computes the entries $v.table[r].Size$, for $r \in [s]$, such that $|\hat{S}_v^{[r]}|$ is maximum and $diam(\hat{T}_v^{[r]}) \leq s$. We analyse each of the four cases for r.

- **Case 0:** In this case, $r = 0$ and $\hat{S}_v^{[0]} = v$. Since this set is the only possible solution for this case, $|\hat{S}_v^{[0]}|$ is maximum and $|\hat{S}_v^{[0]}| = 1$, by Equation 2. Moreover, $diam(\hat{T}_v^{[0]}) = 0 < s$.

- **Case 1:** $1 \leq r \leq \left\lfloor \frac{s}{2} \right\rfloor$. The construction of $\hat{S}_v^{[r]}$, through Equation 3, states that this set consists of v and all its descendant vertices up to a distance of r. Note that $|\hat{S}_v^{[r]}|$ is maximum since $\hat{T}_v^{[r]}$ is the largest tree rooted at v with a height of at most r; furthermore, $diam(\hat{T}_v^{[r]}) = 2r \leq s$.

- **Case 2:** $\left\lfloor \frac{s}{2} \right\rfloor < r \leq s - 1$. This case partitions C_v into u^* and $\{C_v \setminus u^*\}$. Equation 5 states that $\hat{T}_v^{[r]}$ consists of the union of vertex v, tree $\hat{T}_{u^*}^{[r-1]}$, and trees $\hat{T}_{u_i}^{[s-r-1]}$, for each vertex $u_i \in \{C_v \setminus u^*\}$. By the induction hypothesis, $diam(\hat{T}_{u^*}^{[r-1]}) \leq s$ and also $diam(\hat{T}_{u_i}^{[r-1]}) \leq s$, for each $u_i \in \{C_v \setminus u^*\}$. The distance from each vertex $w \in \hat{T}_{u^*}^{[r-1]}$ to v is at most r, and the distance from each vertex $z \in \hat{T}_{u_i}^{[s-r-1]}$ to v is at most $s - r$, for each $u_i \in \{C_v \setminus u^*\}$; therefore, the distance between vertices w and z is at most s; this implies that $diam(\hat{T}_v^{[r]}) \leq s$. The 'max' function in Equation 5 guarantees that $|\hat{S}_v^{[r]}|$ is maximum.

- **Case 3:** $r = s$. Similar to Case 2, the algorithm selects only one vertex $u^* \in C_v$, and $\hat{T}_v^{[s]}$ consists of the union of vertex v and tree $\hat{T}_{u^*}^{[s-1]}$. By the induction hypothesis, $diam(\hat{T}_{u^*}^{[s-1]}) \leq s$ and also $|\hat{S}_{u^*}^{[s-1]}|$ is maximum. Finally, the distance from each vertex $w \in \hat{T}_{u^*}^{[s-1]}$ to v is at most s. This implies that $diam(\hat{T}_v^{[s]}) \leq s$. The 'max' function in Equation 7 guarantees that $|\hat{S}_v^{[s]}|$ is maximum.

□

Corollary 5.2. Upward-sweep *computes the cardinality of a maximum s-club $\hat{S}$ on a rooted tree T.*

Proof. It is enough to extract the entry with the maximum value among all $v.table[r].Size$, for all $v \in V_T$ and $r \in [s]$. □

Lemma 5.3. Downward-sweep *correctly identifies the vertices from T that belong to $\hat{S}$.*

Proof. From Corollary 5.2, the Upward-sweep algorithm correctly computes the cardinality of a maximum s-club $\hat{S}$. Downward-sweep identifies the vertices in $\hat{S}$ by gradually retrieving the information stored at some v's table. Since Downward-sweep only uses information previously computed by Upward-sweep, Downward-sweep is correct. □

5.2 Analysis of DPsC

The following lemmas and theorem prove that DPsC runs in $O(s \cdot n)$ time.

Lemma 5.4. Upward-sweep *takes $O(s \cdot |C_v|)$ time to fill in the table of each vertex v of T.*

Proof. If v is a leaf vertex, Upward-sweep fills in its table in $O(s)$ time, using Equation 1. For internal vertices, we analyse the four cases described in Subsection 4.2.

- **Case 0:** Filling in the $v.table[0].Size$ entry requires $O(1)$ time, by Equation 2.

- **Case 1:** Filling in each $v.table[r].Size$ entry requires $O(|C_v|)$ time, by Equation 3.

- **Case 2:** Filling in both $v.table[r].Size$ and $v.table[r].Pointer$ entries require $O(|C_v|)$ time, by Equations 4-6.

- **Case 3:** Filling in both $v.table[s].Size$ and $v.table[s].Pointer$ entries require $O(|C_v|)$ time, by Equations 7-8.

Overall, Upward-sweep fills in the table of vertex v in $O(s \cdot |C_v|)$ time. □

Lemma 5.5. *The execution time of* Upward-sweep *is* $O(s \cdot n)$.

Proof. Note that $\sum_{\forall v \in V_T} |C_v| = n - 1$; therefore, from Lemma 5.4, Upward-sweep takes $O(s \cdot n)$ time. □

Lemma 5.6. *The execution time of* Downward-sweep *is* $O(n)$.

Proof. Regardless of the value of s, Downward-sweep visits each vertex at most a constant number of times; therefore, such an algorithm requires $O(n)$ time. □

Theorem 5.7. *Let $T = (V_T, E_T)$ be an n-vertex undirected rooted tree. Then,* DPsC *computes a maximum s-club on T in $O(s \cdot n)$ time.*

Proof. It follows from Lemmas 5.5-5.6. □

6 Implementation and experiments

This section presents our experimental design and its results. We compare the performance of DPsC with Schäfer's algorithm, which (to the best of our knowledge) is currently the only known algorithm specifically designed to solve the maximum s-club problem on trees[1].

6.1 Experimental design

- **Input data sets:**

  1. The $\mathcal{T}_{22\_16}$ subset from Professor Brendan McKay[2]. We chose the 'tree22.16.txt' data set containing 12 761 trees; each graph has 22 vertices and a diameter of 16. The experiments used values of $s \in \{8, 9, 10, \ldots, 16\}$. The cumulative execution time of such a set under each algorithm appears in Table 1, Row 1.
  2. $\mathcal{T}_{Ph}$ is a subset of phylogenetic trees chosen from the PhylomeDB [27] catalogue[3]. The chosen source file[4] contains 5 722 phylogenetic trees[5], the largest tree having 297 vertices, and diameters ranging from 2 to 64. The experiments used this set with values of s from the set $\{2, 3, 4, 7, 8, 13, 16, 25, 32, 49, 64\}$. The cumulative execution time of such a set under each algorithm appears in Table 1, Row 14.
  3. T_D is a doubly logarithmic tree [29] with height 5 ($|V_{T_D}| = 119\,041$) and diameter 10. The experiments used this graph with values of $s = \{4, 5\}$. See Table 1, Rows 2 and 3.

[1]Source code and data set are available at: https://alexfloreslamas.github.io/maximum-s-club-problem-on-trees.io/

[2]https://users.cecs.anu.edu.au/~bdm/

[3]Accessible via FTP: http://phylomedb.org/download

[4]phylomones/phylomone_0003/best_trees.txt.gz

[5]"S. cerevisiae phylome made from 60 completely sequenced fungal species"; extracted from: phylomones/-phylomone_0003/phylome_info.txt.gz

Table 1: Wall-clock running time of Schäfer's implementation and DPsC on the six cases of studies. We denote 'timeouts' and 'not applicable' by '—' and 'n/a', respectively.

Row	**Graph**	**Running time in seconds**		$\frac{t_{\text{Schäfer}}}{t_{\text{DP}s\text{C}}}$	**Row**	**Graph**	**Running time in seconds**		$\frac{t_{\text{Schäfer}}}{t_{\text{DP}s\text{C}}}$
		$t_{\text{Schäfer}}$	$t_{\text{DP}s\text{C}}$				$t_{\text{Schäfer}}$	$t_{\text{DP}s\text{C}}$	
1	$\mathcal{T}_{22\_16}$	132.4492	60.8545	2.1	14	$\mathcal{T}_{Ph}$	719.4773	185.4435	3.8
2	$T_D,\ s=4$	109.9832	1.8577	59.2	15	$T_{0.3\text{e}6},\ s=10$	1 022.6843	9.4978	107.7
3	$T_D,\ s=5$	116.5945	1.5734	74.1	16	$T_{0.3\text{e}6},\ s=100$	1 439.42	17.3046	83.1
4	$T_L,\ s=10$	1.5289	0.5584	2.7	17	$T_{0.3\text{e}6},\ s=500$	5 989.8246	74.7689	80.1
5	$T_L,\ s=100$	12.6688	4.8969	2.5	18	$T_{0.5\text{e}6},\ s=10$	2 945.4381	15.6059	188.7
6	$T_L,\ s=1\,000$	602.9626	45.2492	13.3	19	$T_{0.5\text{e}6},\ s=100$	3 654.6010	44.1117	82.8
7	$T_L,\ s=10\,000$	19 859.2017	241.7339	82.1	20	$T_{0.5\text{e}6},\ s=500$	—	2 661.9666	n/a
8	$T_B,\ s=5$	117.9299	2.0445	57.6	21	$T_{0.75\text{e}6},\ s=10$	6 436.4398	22.7611	282.7
9	$T_B,\ s=10$	121.7708	1.9870	61.2	22	$T_{0.75\text{e}6},\ s=100$	7 807.3684	94.7391	82.4
10	$T_B,\ s=15$	125.8261	1.9811	63.5	23	$T_{0.75\text{e}6},\ s=500$	—	5 657.6981	n/a
11	$T_B,\ s=20$	125.4565	1.9592	64.0	24	$T_{1\text{e}6},\ s=10$	11 529.7250	31.8786	361.6
12	$T_B,\ s=25$	126.7537	1.9809	63.9	25	$T_{1\text{e}6},\ s=100$	14 167.638	198.9311	71.2
13	$T_B,\ s=30$	148.4265	1.9831	74.8	26	$T_{1\text{e}6},\ s=500$	—	11 422.2738	n/a

4. T_L is a linear tree with 10 000 vertices. We could think of T_L as an 'opposite' of T_D since T_L is a tall tree and each level has precisely one vertex. The experiments used this graph with values of s from the set $\{10, 100, 1\,000, 10\,000\}$. See Table 1, Rows 4 to 7.

5. T_B is a full-balanced binary tree with 2^{16} leaves. Its diameter is 32. The experiments used this tree with values of $s = \{5,\ 10,\ 15,\ 20,\ 25,\ 30\}$. See Table 1, Rows 8 to 13.

6. The $T_{\eta\text{e}6}$ trees, for $\eta \in \{0.3, 0.5, 0.75, 1\}$ and e6 $= 10^6$, have 0.3×10^6, 0.5×10^6, 0.75×10^6, and 1×10^6 vertices. The diameters of these graphs are 2311, 1850, 2729, and 3375, respectively. We built these trees with the instruction 'nx.random_tree(...)' from NetworkX [20]. We used the following values of $s = \{10, 100, 500\}$ for each tree. See Table 1, Rows 15 to 26.

- **Evaluation metrics**. We measured the wall-clock running time of both Schäfer's algorithm and DPsC implementations. We focused on the time taken to compute the cardinality of a maximum s-club for each input graph, excluding the time for operations such as reading/writing files, initialising data structures, and identifying the vertices in a maximum s-club. Since both algorithms are correct, we did not report the cardinalities of the computed maximum s-clubs.

- **Computing environment.** We implemented both algorithms in Python 3.10 [49] and the Python package NetworkX [20] version 2.8. The input graphs are in GML [26] format. The experiments ran on a 2012 MacBook Pro with a 2.3 GHz Quad-Core Intel Core i7 processor, running a 64-bit macOS Catalina version 10.15.7 with a total memory of 8 GB (1600 MHz DDR3). This computer executed each algorithm implementation on each input graph sequentially without resource allocation.

Remark 6.1. *Similarly to Schäfer's algorithm (and implementation), our implementation of* DPsC *fills in as many rows in each vertex's table as the height of each subtree rooted in such a vertex (Pseudocode 2 does not show this modification).*

6.2 Experimental results

Table 1 presents the results of our experiments. Rows 1 and 14 show the total execution time for each set. Columns 5 and 10 present the ratio between $t_{\text{Schäfer's}}$ and $t_{\text{DP}s\text{C}}$. The results of the experiments show that DPsC outperforms Schäfer's algorithm in all cases. For the case of studies $\mathcal{T}_{22\_16}$ (Row 1) and $\mathcal{T}_{Ph}$ (Row 14), DPsC is between 2 to 4 times faster than Schäfer's algorithm. We conjecture that this behaviour occurs because those data sets only consist of small graphs. For the cases of large trees, T_D (Rows 2 and 3) and T_B (Rows 8-13), DPsC is between 57 to 75 times faster than Schäfer's algorithm. In the case of the large tree T_L (Rows 4-7), the wall-clock running times of the algorithms depend heavily on the value of s, with a greater ratio of $t_{\text{Schäfer}}$ to $t_{\text{DP}s\text{C}}$ as the value of s increases.

Finally, for the case of study $T_{\eta\text{e}6}$ (Rows 15-26), DPsC shows a speedup of at least 70 over Schäfer's algorithm for some rows. We anticipated this favouring results for DPsC given Schäfer's $O(s)$ overhead. Due to time constraints, we only reported the running time of Schäfer's implementation for values of $s \in \{10, 100\}$.

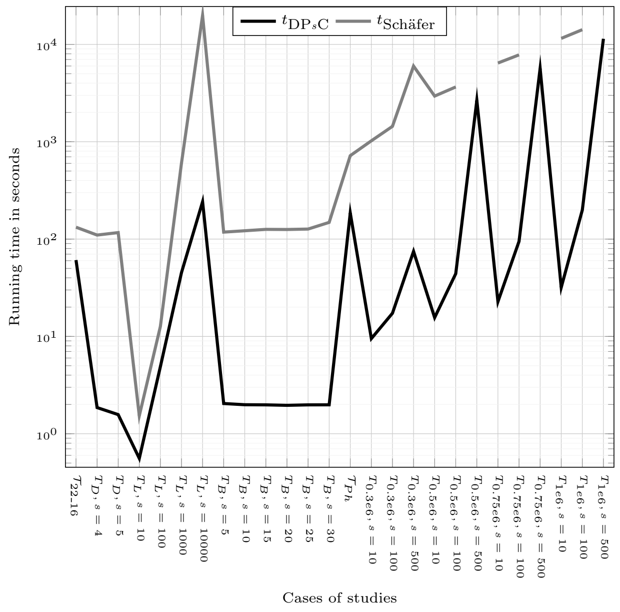


Figure 5: Execution time comparison between Schäfer's algorithm and DPsC using a logarithmic scale.

Similarly to Table 1, Figure 5 displays a line chart comparing the wall-clock running time of Schäfer's algorithm and DPsC to compute a maximum s-club on the six input data sets. The dark grey lines represent Schäfer's algorithm ($t_{\text{Schäfer}}$) and the bold line DPsC ($t_{\text{DP}s\text{C}}$). The y-axis, presented on a logarithmic scale, displays the running time in seconds required by the algorithms. The x-axis shows each tree in the dataset and the s value for the maximum s-club. We omit the s values for case studies $\mathcal{T}_{22\_16}$ and $\mathcal{T}_{Ph}$ (please refer to Section 6.1 for details.) The gaps in $t_{\text{Schäfer}}$ in case studies $T_{0.5\text{e}6}$, $T_{0.75\text{e}6}$, and $T_{1\text{e}6}$ denote timeouts when $s = 500$. The chart shows that DPsC outperforms Schäfer's algorithm in all study cases. These results support the theoretical complexity analysis and suggest that DPsC may be more efficient than Schäfer's algorithm in practice.

Example 6.2. *Figure 6 shows the result of executing* DPsC *on a specific phylogenetic tree from the set* $\mathcal{T}_{Ph}$*. The grey subgraph represents a maximum* 13*-club of cardinality* 50*.*

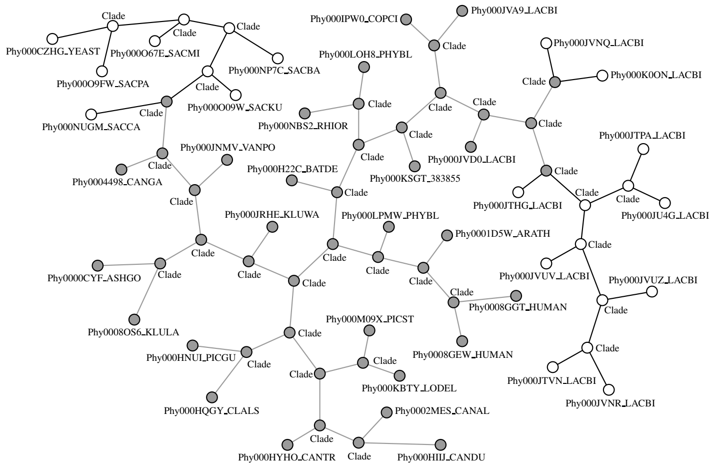


Figure 6: A phylogenetic tree $T \in \mathcal{T}_{Ph}$ with 74 vertices and a diameter of 23; grey vertices belong to a maximum 13-club.

7 Conclusion and future direction

This paper presents DPsC, an efficient algorithm to solve the maximum s-club problem in an arbitrary tree with n vertices for any positive integer s. The algorithm consists of two phases: first, computing the cardinality of a maximum s-club, and second, identifying the vertices in such a set. We proved the correctness of DPsC, analysed its time complexity, and performed experimental simulations on six graph data sets. The algorithm has a time complexity of $O(s{\cdot}n)$. Theoretical and experimental results show that the running time of DPsC outperforms the best previous algorithm for the same problem.

Our approach intentionally balances a simple methodology with practical efficiency. While the algorithm may appear straightforward, this simplicity is a significant advantage. It allows for more accessible implementation, easier integration into educational settings, and, as demonstrated by our results, superior performance compared to the existing algorithm.

Although our proposal has a better asymptotic time complexity, practical considerations such as constant factors in Python's overhead, memory efficiency, and experimental setup can explain why the reported speed-up is not directly proportional to the theoretical difference between s and s^2. Running the algorithms on larger inputs or in an optimised environment could be necessary to observe a clearer distinction.

For future work, we aim to explore the possibility of extending this dynamic programming approach to broader graph families with a tree-like structure.

Acknowledgements

Funding: A. Flores-Lamas and M. Hague are supported by the Engineering and Physical Sciences Research Council, EPSRC [EP/T00021X/1]. J. A. Trejo-Sánchez is supported by the National Council of Science and Technology, Mexico (SECIHTI), Frontier Science Project No. 304006.

We thank the anonymous reviewers for their thoughtful comments and suggestions, which helped improve the clarity and quality of this manuscript.

References

[1] Arkhat Abzhanov. Phylogenetic analysis and it's applications. *Journal of Phylogenetics & Evolutionary Biology*, 9(8):175, 2021.

[2] Richard D Alba. A graph-theoretic definition of a sociometric clique. *Journal of Mathematical Sociology*, 3(1):113–126, 1973.

[3] Maria Teresa Almeida and Filipa D Carvalho. Integer models and upper bounds for the 3-club problem. *Networks*, 60(3):155–166, 2012.

[4] Yuichi Asahiro, Eiji Miyano, and Kazuaki Samizo. Approximating maximum diameter-bounded subgraphs. In *Latin American Symposium on Theoretical Informatics*, pages 615–626. Springer, 2010.

[5] Balabhaskar Balasundaram. *Graph theoretic generalizations of clique: Optimization and extensions.* PhD thesis, Texas A&M University, 2007.

[6] Balabhaskar Balasundaram, Sergiy Butenko, and Svyatoslav Trukhanov. Novel approaches for analyzing biological networks. *Journal of Combinatorial Optimization*, 10(1):23–39, 2005.

[7] Balabhaskar Balasundaram and Foad Mahdavi Pajouh. *Graph Theoretic Clique Relaxations and Applications*, pages 1559–1598. Springer New York, New York, NY, 2013.

[8] Ambroise Baril, Antoine Castillon, and Nacim Oijid. On the parameterized complexity of non-hereditary relaxations of clique. *Theoretical Computer Science*, 1003:114625, 2024.

[9] Jean-Marie Bourjolly, Gilbert Laporte, and Gilles Pesant. Heuristics for finding k-clubs in an undirected graph. *Computers & Operations Research*, 27(6):559–569, 2000.

[10] Jean-Marie Bourjolly, Gilbert Laporte, and Gilles Pesant. An exact algorithm for the maximum k-club problem in an undirected graph. *European Journal of Operational Research*, 138(1):21–28, 2002.

[11] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[12] Filipa D Carvalho and M Teresa Almeida. Upper bounds and heuristics for the 2-club problem. *European Journal of Operational Research*, 210(3):489–494, 2011.

[13] Parinya Chalermsook, Marek Cygan, Guy Kortsarz, Bundit Laekhanukit, Pasin Manurangsi, Danupon Nanongkai, and Luca Trevisan. From gap-eth to fpt-inapproximability: Clique, dominating set, and more. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 743–754. IEEE, 2017.

[14] Riccardo Dondi, Giancarlo Mauri, Florian Sikora, and Zoppis Italo. Covering a graph with clubs. *Journal of Graph Algorithms and Applications*, 23(2), 2019.

[15] Uriel Feige. Approximating maximum clique by removing subgraphs. *SIAM Journal on Discrete Mathematics*, 18(2):219–225, 2004.

[16] Joseph Felsenstein. Distance methods for inferring phylogenies: a justification. *Evolution*, pages 16–24, 1984.

[17] Alejandro Flores-Lamas, José Alberto Fernández-Zepeda, and Joel Antonio Trejo-Sánchez. Algorithm to find a maximum 2-packing set in a cactus. *Theoretical Computer Science*, 725:31–51, 2018.

[18] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[19] Timo Gschwind, Stefan Irnich, Fabio Furini, Roberto Wolfler Calvo, et al. Social network analysis and community detection by decomposing a graph into relaxed cliques. Technical report, Gutenberg School of Management and Economics, Johannes Gutenberg-Universität Mainz, 2015.

[20] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[21] Eric Harley, Anthony Bonner, and Nathan Goodman. Uniform integration of genome mapping data using intersection graphs. *Bioinformatics*, 17(6):487–494, 2001.

[22] Juris Hartmanis. Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson). *Siam Review*, 24(1):90, 1982.

[23] Sepp Hartung, Christian Komusiewicz, André Nichterlein, and Ondřej Suchỳ. On structural parameterizations for the 2-club problem. *Discrete Applied Mathematics*, 185:79–92, 2015.

[24] Sepp Hartung, Christian Komusiewicz, and André Nichterlein. Parameterized algorithmics and computational experiments for finding 2-clubs. *Journal of Graph Algorithms and Applications*, 19(1):155–190, Jan. 2015.

[25] Johan Håstad. Clique is hard to approximate within $n^{1-\varepsilon}$. *Acta Mathematica*, 182:105–142, 1999.

[26] Michael Himsolt. Gml: A portable graph file format. Technical report, Technical report, Universitat Passau, 1997.

[27] Jaime Huerta-Cepas, Salvador Capella-Gutiérrez, Leszek P Pryszcz, Marina Marcet-Houben, and Toni Gabaldón. Phylomedb v4: zooming into the plurality of evolutionary histories of a genome. *Nucleic acids research*, 42(D1):D897–D902, 2014.

[28] Iker Irisarri, Denis Baurain, Henner Brinkmann, Frédéric Delsuc, Jean-Yves Sire, Alexander Kupfer, Jörn Petersen, Michael Jarek, Axel Meyer, Miguel Vences, et al. Phylotranscriptomic consolidation of the jawed vertebrate timetree. *Nature ecology & evolution*, 1(9):1370–1378, 2017.

[29] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., USA, 1992.

[30] Steven Laan, Maarten Marx, and Robert J Mokken. Close communities in social networks: boroughs and 2-clubs. *Social Network Analysis and Mining*, 6:1–16, 2016.

[31] Gayatri Mahapatro, Debahuti Mishra, Kailash Shaw, Sashikala Mishra, and Tanushree Jena. Phylogenetic tree construction for dna sequences using clustering methods. *Procedia Engineering*, 38:1362–1366, 2012.

[32] Raffaele Marino, Lorenzo Buffoni, and Bogdan Zavalnij. A Short Review on Novel Approaches for Maximum Clique Problem: From Classical algorithms to Graph Neural Networks and Quantum algorithms, March 2024.

[33] S Mohseni-Zadeh, Pierre Brézellec, and J-L Risler. Cluster-c, an algorithm for the large-scale clustering of protein sequences based on the extraction of maximal cliques. *Computational Biology and Chemistry*, 28(3):211–218, 2004.

[34] Robert J Mokken. *Cliques, clubs and clans*, volume 6. Klett-Cotta, 1980.

[35] Esmaeel Moradi and Balabhaskar Balasundaram. Finding a maximum k-club using the k-clique formulation and canonical hypercube cuts. *Optimization Letters*, 12(8):1947–1957, 2018.

[36] L. Nakhleh. Evolutionary trees. In Stanley Maloy and Kelly Hughes, editors, *Brenner's Encyclopedia of Genetics (Second Edition)*, pages 549–550. Academic Press, San Diego, second edition edition, 2013.

[37] Patric RJ Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1-3):197–207, 2002.

[38] F Mahdavi Pajouh and Balabhaskar Balasundaram. On inclusionwise maximal and maximum cardinality k-clubs in graphs. *Discrete Optimization*, 9(2):84–97, 2012.

[39] Foad Mahdavi Pajouh, Balabhaskar Balasundaram, and Illya V Hicks. On the 2-club polytope of graphs. *Operations Research*, 64(6):1466–1481, 2016.

[40] Hao Pan, Yajun Lu, Balabhaskar Balasundaram, and Juan S Borrero. Finding conserved low-diameter subgraphs in social and biological networks. *Networks*, 84(4):509–527, 2024.

[41] Jeffrey Pattillo, Nataly Youssef, and Sergiy Butenko. Clique relaxation models in social network

analysis. In My T. Thai and Panos M. Pardalos, editors, *Handbook of Optimization in Complex Networks: Communication and Social Networks*, chapter 5, pages 143–162. Springer New York, New York, NY, 2012.

[42] Manon Ragonnet-Cronin, Emma Hodcroft, Stéphane Hué, Esther Fearnhill, Valerie Delpech, Andrew J Leigh Brown, and Samantha Lycett. Automated analysis of phylogenetic clusters. *BMC bioinformatics*, 14:1–10, 2013.

[43] Ali Rahim Taleqani, Chrysafis Vogiatzis, and Jill Hough. Maximum closeness centrality k-clubs: A study of dock-less bike sharing. *Journal of Advanced Transportation*, 2020(1):1275851, 2020.

[44] Marc Sageman. *Understanding terror networks*. University of Pennsylvania press, 2004.

[45] Hosseinali Salemi and Austin Buchanan. Parsimonious formulations for low-diameter clusters. *Mathematical Programming Computation*, 12(3):493–528, 2020.

[46] Alexander Schäfer. *Exact algorithms for s-club finding and related problems*. PhD thesis, Friedrich-Schiller-University Jena, 2009.

[47] Alexander Schäfer, Christian Komusiewicz, Hannes Moser, and Rolf Niedermeier. Parameterized computational complexity of finding small-diameter subgraphs. *Optimization Letters*, 6:883–891, 2012.

[48] Shahram Shahinpour and Sergiy Butenko. Algorithms for the maximum k-club problem in graphs. *Journal of Combinatorial Optimization*, 26(3):520–554, 2013.

[49] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[50] Alexander Veremyev and Vladimir Boginski. Identifying large robust network clusters via new compact formulations of maximum k-club problems. *European Journal of Operational Research*, 218(2):316–326, 2012.

[51] Alexander Veremyev, Vladimir Boginski, Eduardo L Pasiliao, and Oleg A Prokopyev. On integer programming models for the maximum 2-club problem and its robust generalizations in sparse graphs. *European Journal of Operational Research*, 297(1):86–101, 2022.

[52] Alexander Veremyev, Oleg A Prokopyev, and Eduardo L Pasiliao. Critical nodes for distance-based connectivity and related problems in graphs. *Networks*, 66(3):170–195, 2015.

[53] Qinghua Wu and Jin-Kao Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, 2015.

[54] Oleksandra Yezerska, Foad Mahdavi Pajouh, Alexander Veremyev, and Sergiy Butenko. Exact algorithms for the minimum s-club partitioning problem. *Annals of Operations Research*, 276:267–291, 2019.

[55] Oleksandra Yezerska, Foad Mahdavi Pajouh, and Sergiy Butenko. On biconnected and fragile subgraphs of low diameter. *European Journal of Operational Research*, 263(2):390–400, 2017.