Searching Cycle-Disjoint Graphs

Boting Yang, Runtao Zhang, and Yi Cao
Technical Report TR-CS 2006-5
March, 2006

Department of Computer Science
University of Regina
Regina, Saskatchewan, CANADA
S4S 0A2

ISBN 0-7731-0571-9 (print)
ISBN 0-7731-0572-7 (on-line)

Searching Cycle-Disjoint Graphs*

Boting Yang, Runtao Zhang, Yi Cao
Department of Computer Science
University of Regina

February 26, 2006

Abstract. In this paper, we consider the edge searching problem on cycle-disjoint graphs. We first improve the running time of the algorithm to compute the vertex separation and the optimal layout of a unicyclic graph, which is given by Ellis et al. (2004), from $O(n \log n)$ to $O(n)$. By a linear-time transformation, we can compute the edge search number of a unicyclic graph in linear time. We also propose an $O(n)$ time algorithm to compute the edge search number and the optimal edge search strategy of a cycle-disjoint graph in which every cycle has at most three vertices with degree more than two. We show how to compute the search number for a k-ary cycle-disjoint graph. We also present some results on approximation algorithms.

Keywords: edge searching, vertex separation, cycle-disjoint graph, unicyclic graph.

1 Introduction

The edge searching problem is to find the minimum number of searchers to capture an intruder that is hiding on vertices or edges of a graph [10]. There are other searching models besides edge searching, but in this paper we mainly consider the edge searching problem. For this reason, we will use "search" instead of "edge search" for simplicity.

Let G be a graph without loops and multiple edges. Initially, all vertices and edges of G are *contaminated*, which means an intruder can hide on any vertices or anywhere along edges. There are three *actions* for searchers: (1) place a searcher on a vertex; (2) remove a searcher from a vertex; (3) slide a searcher along an edge from one end vertex to the other. A *search strategy* is a sequence of actions designed so that the final action leaves all edges of G cleared. An edge uv in G can be *cleared* in one of the following two ways by a sliding action: (1) two searchers are located on vertex u, and one of them slides along uv from u to v; or (2) a searcher is located on vertex u, where all edges incident with u, other than uv, are already cleared, and the searcher slides from u to v. The intruder can slide along a path that contains no searcher at a great speed at any time. The minimum number of searchers required to clear G is called the *search number* of G, denoted by $s(G)$. A search strategy for G is *optimal* if this strategy clears G using $s(G)$ searchers.

Let S be a search strategy for a graph G and let $E(i)$ be the set of cleared edges just after action i. S is said to be *monotonic* if $E(i) \subseteq E(i+1)$ for each i. LaPaugh [11] proved that for any connected graph G, allowing recontamination cannot reduce the search number. Thus, throughout this paper, we only need to consider monotonic search strategies. For this reason, we will use "search strategy" instead of "monotonic search strategy" for simplicity.

Megiddo et al. [12] showed that determining the search number of a graph G is NP-complete. They also gave an $O(n)$ time algorithm to compute the search number of a tree and an $O(n \log n)$ time algorithm to find the optimal search strategy, where n is the number of vertices in the tree. Peng et al. [14] proposed an $O(n)$ time algorithm to compute the optimal search strategy of a tree.

Search numbers are closely related to several other important graph parameters. Ellis et al. [6] proved that for any connected undirected graph G and its 2-expansion G', the vertex operation of G' equals the search number of G', which has the same search number as G. Kinnersley [9] showed that vertex separation is identical to pathwidth, an important measure of graph structure.

A *layout* of a connected graph $G(V, E)$ is a one to one mapping L: $V \to \{1, 2, \ldots, |V|\}$. Let $V_L(i) = \{x : x \in V(G)$, and there exists $y \in V(G)$ such that the edge $xy \in E(G)$, $L(x) \le i$ and $L(y) > i\}$. The *vertex separation of* G *with respect to* L, denoted by $\mathrm{vs}_L(G)$, is defined as $\mathrm{vs}_L(G) = \max\{|V_L(i)| : 1 \le i \le |V(G)|\}$. The *vertex separation of* G is defined as $\mathrm{vs}(G) = \min\{\mathrm{vs}_L(G) : L$ is a layout of $G\}$. We say that L is an *optimal layout* if $\mathrm{vs}_L(G) = \mathrm{vs}(G)$. Ellis et al. [6] proved that $\mathrm{vs}(G) \le s(G) \le \mathrm{vs}(G) + 2$ for any connected

*This research was supported in part by NSERC and MITACS.

undirected graph G. They also proposed an algorithm to compute the vertex separation of a tree in $O(n)$ time. Based on this algorithm, Ellis and Markov [7] gave an $O(n \log n)$ time algorithm to compute the vertex separation and the corresponding optimal layout of a unicyclic graph.

Bodlaender and Kloks [3] gave a polynomial time algorithm for computing the pathwidth of a graph with constant treewidth. Since the search number of a graph equals the pathwidth of its 2-expansion, we know that the search number of a graph with constant treewidth is polynomial time computable. However, the exponent in the running time of this algorithm is very large. Even for a graph with treewidth two, it takes $\Omega(n^{11})$ time. Bodlaender and Fomin [4] introduced an $O(n)$ time approximation algorithm to compute the pathwidth of an outerplanar graph, a class of graphs with treewidth two. The approximation ratio of their algorithm is 2. Finding efficient algorithms for computing the search number of a graph with constant treewidth continues to be a challenge.

All graphs in this paper are finite without loops and multiple edges. A graph G is called a *cycle-disjoint graph (CDG)* if it is connected and no pair of cycles in G share a vertex. If every cycle of a CDG G has at most three vertices with degree more than two, then we call G a *3-cycle-disjoint graph (3CDG)*. If a vertex or an edge is on a cycle of G, it is called, respectively, a *cycle vertex* or a *cycle edge*.

Our motivation is to find an efficient algorithm for computing the search number of a graph with treewidth at most two. We have successfully found an $O(n)$ time algorithm for a unicyclic graph. Then we tried to extend this algorithm to CDGs. However, we found the necessary structural information of CDGs is much more complicated than that of unicyclic graphs. Finally, we managed to develop an $O(n)$ time algorithm for 3CDGs. We also found the search numbers of k-ary CDGs that is a class of CDGs with well balanced structures.

This paper is organized as follows. In Section 2, we improve Ellis and Markov's algorithm in [7] from $O(n \log n)$ to $O(n)$. In Section 3, we propose a linear time algorithm to compute the search number and the optimal search strategy of a 3-cycle-disjoint graph using the labeling method. In Section 4, we show how to compute the search number of a k-ary cycle-disjoint graph. In Section 5, we investigate approximation algorithms, and finally in Section 6, we discuss issues arising from these results.

2 Unicyclic graphs

Ellis and Markov [7] proposed an $O(n \log n)$ algorithm to compute the vertex separation and the optimal layout of a unicyclic graph using the labeling method. In this section we will give an improved algorithm that can do the same work in $O(n)$ time. All definitions and notation in this section are from [7]. Their algorithm consists of three functions: main, vs_uni and vs_reduced_uni (see Fig. 28, 29 and 30 in [7] for their descriptions).

Let U be a unicyclic graph and e be a cycle edge of U. The function main, first computes the vertex separation of the tree $U - e$, and then invokes function vs_uni to decide whether $\text{vs}(U) = \text{vs}(U - e)$. vs_uni is a recursive function that has $O(\log n)$ depth, and in each iteration it computes the vertex separation of a reduced tree $U' - e$ and this takes $O(n)$ time. Thus, the running time of vs_uni is $O(n \log n)$. vs_uni may invoke the function vs_reduced_uni to decide whether a unicyclic graph U is k-conforming. vs_reduced_uni is also a recursive function that has $O(\log n)$ depth, and in each iteration it may compute the vertex separation of $T_1[a]$ and $T_1[b]$ and this takes $O(n)$ time. Hence, the running time of vs_reduced_uni is also $O(n \log n)$.

We will modify all three functions. The main improvement of our algorithm is to preprocess the input of both vs_uni and vs_reduced_uni such that we can achieve $O(n)$ time. Refer to [7] for the linear time algorithm to compute the vertex separation and the optimal layout of a tree. The following is our improved algorithm, which computes the vertex separation and the optimal layout of a unicyclic graph U.

program main_modified
1 For each constituent tree, compute its vertex separation, optimal layout and type.
2 Arbitrarily select a cycle edge e and a cycle vertex r. Let $T[r]$ denote $U - e$ with root r.
  Compute $\text{vs}(T[r])$ and the corresponding layout X.
3 Let L be the label of r in $T[r]$. Set $\alpha \leftarrow \text{vs}(T[r])$, $k \leftarrow \text{vs}(T[r])$.
4 **while** the first element of L is a k-critical element and the corresponding k-critical vertex v
    is not a cycle vertex in U, **do**
      Update U by deleting $T[v]$ and update L by deleting its first element;
      Update the constituent tree $T[u]$ that contains v by deleting $T[v]$
        and update the label of u in $T[u]$ by deleting its first element;
      $k \leftarrow k - 1$;
5 **if** (vs_uni_modified(U, k))
    then output(α, the layout created by vs_uni_modified);
    else output($\alpha + 1$, X);

function vs_uni_modified(U, k): Boolean
Case 1: U has one k-critical constituent tree;
compute vs(T');
if vs$(T') = k$, **then return** (false) **else return** (true);
Case 2: U has three or more non-critical k-trees;
return (false);
Case 3: U has exactly two non-critical k-trees T_i and T_j;
compute vs$(T_1[a])$, vs$(T_1[b])$, vs$(T_2[c])$ and vs$(T_2[d])$;
/* Assume that vs$(T_1) \geq$ vs(T_2). */
/* Let L_a be the label of a in $T_1[a]$, and L_b be the label of b in $T_1[b]$. */
/* Let L_c be the label of c in $T_2[c]$, and L_d be the label of d in $T_2[d]$. */
/* Let U' be U minus the bodies of T_i and T_j. */
return (vs_reduced_uni_modified$(U', L_a, L_b, L_c, L_d, k)$);
Case 4: U has exactly one non-critical k-tree T_i;
/* let q be the number of $(k-1)$-trees that is not type NC. */
Case 4.1: $0 \leq q \leq 1$;
return (true);
Case 4.2: $q = 2$;
for each tree T_j from among the two $(k-1)$-trees, **do**
compute the corresponding vs$(T_1[a])$, vs$(T_1[b])$, vs$(T_2[c])$ and vs$(T_2[d])$;
if (vs_reduced_uni_modified$(U', L_a, L_b, L_c, L_d, k)$) **then return** (true);
/* U' is equal to U minus the bodies of T_i and T_j. */
return (false);
Case 4.3: $q = 3$;
for each tree T_j from among the three $(k-1)$-trees, **do**
compute the corresponding vs$(T_1[a])$, vs$(T_1[b])$, vs$(T_2[c])$ and vs$(T_2[d])$;
if (vs_reduced_uni_modified$(U', L_a, L_b, L_c, L_d, k)$) **then return** (true);
/* U' is equal to U minus the bodies of T_i and T_j. */
return (false);
Case 4.4: $q \geq 4$;
return (false);
Case 5: U has no k-trees;
/* let q be the number of $(k-1)$-trees that is not type NC. */
Case 5.1: $0 \leq q \leq 2$;
return (true);
Case 5.2: $q = 3$;
for each choice of two trees T_i and T_j from among the three $(k-1)$-trees, **do**
compute the corresponding vs$(T_1[a])$, vs$(T_1[b])$, vs$(T_2[c])$ and vs$(T_2[d])$;
if (vs_reduced_uni_modified$(U', L_a, L_b, L_c, L_d, k)$) **then return** (true);
/* U' is equal to U minus the bodies of T_i and T_j. */
return (false);
Case 5.3: $q = 4$;
for each choice of two trees T_i and T_j from among the four $(k-1)$-trees, **do**
compute the corresponding vs$(T_1[a])$, vs$(T_1[b])$, vs$(T_2[c])$ and vs$(T_2[d])$;
if (vs_reduced_uni_modified$(U', L_a, L_b, L_c, L_d, k)$) **then return** (true);
/* U' is equal to U minus the bodies of T_i and T_j. */
return (false);
Case 5.4: $q \geq 5$;
return (false).

function vs_reduced_uni_modified$(U, L_a, L_b, L_c, L_d, k))$: Boolean
/* Let a_1, b_1, c_1, d_1 be the first elements of L_a, L_b, L_c, L_d respectively. */
/* Let $|a_1|, |b_1|, |c_1|, |d_1|$ be the value of a_1, b_1, c_1, d_1 respectively. We assume that $|a_1| \geq |c_1|$. */
Case 1: $|a_1| = k$;
return (false).
Case 2: $|a_1| < k - 1$;
return (true).
Case 3: $|a_1| = k - 1$;
if both a_1 and b_1 are $(k-1)$-critical elements,
then
/* Let u be the $(k-1)$-critical vertex in $T_1[a]$
and let v be the $(k-1)$-critical vertex in $T_1[b]$. */
if $u = v$ and u is not a cycle vertex,
then
update L_a and L_b by deleting their first elements;
update U by deleting $T[u]$;
update the label of the root of the constituent tree containing u by deleting its first element;
if $|c_1|$ is greater than the value of the first element in current L_a,
then return (vs_reduced_uni_modified$(U, L_c, L_d, L_a, L_b, k-1)$.
else return (vs_reduced_uni_modified$(U, L_a, L_b, L_c, L_d, k-1)$.
else /* ($u = v$ and u is a cycle vertex) or ($u \neq v$) */
return (T_2 contains no $k-1$ types other than NC constituents);
else return ((neither a_1 nor d_1 is $(k-1)$-critical element)
or (neither b_1 nor c_1 is $(k-1)$-critical element)).

Lemma 2.1 *Let U be a unicyclic graph, e be a cycle edge and r be a cycle vertex in U. Let $T[r]$ denote the rooted tree $U - e$ with root r. If $\mathrm{vs}(T[r]) = k$, then U has a k-constituent tree of type Cb if and only if the first element in the label of r in $T[r]$ is a k-critical element and the corresponding k-critical vertex is not a cycle vertex.*

PROOF. ($\Rightarrow$). Let T' be the k-constituent tree of type Cb, and u be the only cycle vertex in T' and v be the k-critical vertex in $T'[u]$. It is easy to see that all vertices in T' except u are descendants of u in $T[r]$. After computing the vertex separation of $T[r]$, each vertex in $T[r]$ obtained a label. The first element in the label of u must be a k-critical element and the corresponding k-critical vertex is v. Since $\mathrm{vs}(T[r]) = k$, which means $\mathrm{vs}(T[r]-T[u]) < k$, the first element in the label of r is also a k-critical element and the corresponding k-critical vertex is v, which is not a cycle vertex.

($\Leftarrow$). Let v be the k-critical vertex in $T[r]$ and let T' be the constituent tree that contains v and let u be the only cycle vertex in T'. It is easy to see that v is a descendant of u in $T[r]$. Thus, $T[v]$ is a subgraph of T' and T' is a k-constituent tree of type Cb in U. ■

The correctness of the modified algorithm follows from the analysis in Sections 4 and 5 in [7]. We now compare the two algorithms. In our main_modified function, if the condition of the *while-loop* is satisfied, then by Lemma 2.1, U has a k-constituent tree of type Cb that contains v. Let $T'[u]$ be this constituent tree and u be the only cycle vertex in $T'[u]$. The first element in the label of u in $T'[u]$ must be k-critical element. Let $L(r)$ be the label of r in $T[r]$ and $L(u)$ be the label of u in $T'[u]$. We can obtain the label of r in $T[r] - T[v]$ and the label of u in $T'[u] - T'[v]$ by deleting the first element of each label, according to the definition of labels [7]. This work can be done in constant time. However, without choosing a cycle vertex as the root of T, their algorithm needs $O(n)$ time to compute these two labels. Function vs_uni in [7] can only invoke itself in Case 1 when U has a k-constituent tree of type Cb. Our main_modified function invokes function vs_uni_modified only when the condition of the *while-loop* is not satisfied. By Lemma 2.1, in this case, U does not have a k-constituent tree of type Cb. Thus in Case 1 of vs_uni_modified, the tree must be of type C, and recursion is avoided. In their function vs_reduced_uni, $vs(T_1)$ and $vs(T_2)$ are computed using $O(n)$ time. However, we compute them before invoking vs_reduced_uni_modified. Let L_a, L_b, L_c and L_d be the label of a in $T_1[a]$, b in $T_1[b]$, c in $T_2[c]$ and d in $T_2[d]$ respectively. All the information needed by vs_reduced_uni_modified is these four labels. While recursion occurs, we can obtain new labels by simply deleting the first elements from the old ones, which requires only constant time. Hence, the time complexity of vs_reduced_uni_modified can be reduced to O(1) if we do not count the recursive iterations.

We now analyze the running time of our modified algorithm. Since function vs_reduced_uni_modified only ever invokes itself and the depth of the recursion is $O(\log n)$, its running time is $O(\log n)$. In function vs_uni_modified, Case 1 needs $O(n)$; Cases 3, 4.2, 4.3, 5.2 and 5.3 need $O(n) + O(\log n)$; and other cases can be done in $O(1)$. Thus, the running time of vs_uni_modified is $O(n)+O(\log n)$. In the main_modified function, all the work before invoking vs_uni_modified can be done in $O(n) + O(\log n)$. Therefore, the total running time of the modified algorithm is $O(n)$.

Theorem 2.2 *For a unicyclic graph G, the vertex separation and the optimal layout of G can be computed in linear time.*

For a graph G, the 2-expansion of G is the graph obtained by replacing each edge of G by a path of length three. By Theorem 2.2 in [6], the search number of G is equal to the vertex separation of the 2-expansion of G. From Theorem 2.2, we have the following result.

Corollary 2.3 *For a unicyclic graph G, the search number of G can be computed in linear time.*

3 3-cycle-disjoint graphs

The main work of this section is to propose an $O(n)$ time algorithm to compute the search number and the corresponding optimal search strategy of a 3CDG. In this algorithm we extend the labeling method used in [6]. First of all, we introduce some notation and definitions.

3.1 Notation and definitions

For two graphs G and H, the *union* of G and H, denoted by $G \cup H$, is the graph with vertex set $V(G) \cup V(H)$ and edge set $E(G) \cup E(H)$. If H is a small graph that consists of only one or two edges, we may use $G \cup E(H)$ to represent $G \cup H$.

Recall that a CDG G is a connected graph such that no pair of cycles in G share a vertex. A *rooted CDG* is a connected CDG with one vertex designated as the root of the graph. Let $G[r]$ be a rooted CDG with root r. We first define that each vertex of $G[r]$ except r is a descendant of r. For any edge uv that is not on a cycle, the graph $G[r] - uv$ has two connected components. If $u = r$ or u and r are in the same component, then we say that v is a *child* of u, and each vertex that is in the same component as v is called a *descendant* of u. Note that there is no parent-child relationship among vertices in the same cycle. If v is the child of u, then we orient this edge with the direction from u to v and the oriented edge is denoted by (u, v). For any cycle $v_1 v_2 \ldots v_k v_1$ in $G[r]$, if v_1 has an incoming edge (u, v_1) in $G[r]$ or $v_1 = r$, then v_1 is referred to as the *entrance-vertex* of the cycle. For any vertex v of $G[r]$, the subgraph induced by v and all its descendant vertices is called the *vertex-branch* of v, denoted by $G[v]$. $G[r]$ can be considered as a vertex-branch of r. For any directed edge (u, v) of $G[r]$, let G_v be the connected component of $G[r] - (u, v)$ that contains v. The graph $G_v \cup \{(u, v)\}$ is called the *edge-branch* of the directed edge (u, v), denoted by $G[uv]$. We also say that $G[uv]$ is an edge-branch of u. Edge-branch is only defined for non-cycle edges since only these edges are oriented.

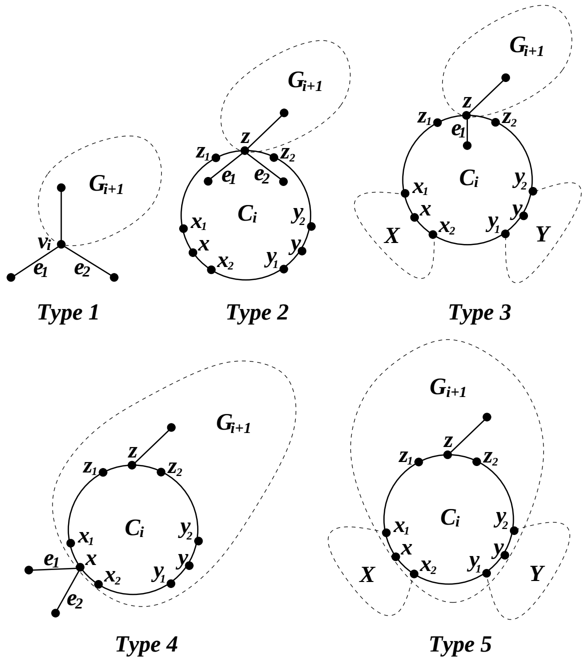

Figure 1: five typical critical structures of G_i

In our labeling process, we assign proper labels to all vertices, non-cycle edges and cycles of a rooted CDG $G[r]$. The label of a vertex v, non-cycle edge e or cycle C in $G[r]$ records the necessary structural information of $G[v]$, $G[e]$ or $G[C]$, respectively. Intuitively, a label is a sequence of elements $(s_1^{t_1}, s_2^{t_2}, \ldots s_m^{t_m})$, where each element $s_i^{t_i}$ consists of a positive integer s_i and a superscript t_i, where $0 \le t_i \le 5$. Let G_1 be $G[v]$ (similarly, $G[e]$ or $G[C]$) and s_1 be the search number of G_1. If G_1 has two edge disjoint subgraphs (they may share a vertex) such that each of them has search number $s(G_1)$, then we say G_1 is *critical* and G_1 must be one of the five typical structures illustrated in Figure 1 and t_1 indicates which structure it is. If G_1 is not critical, then we say G_1 is *non-critical* and $t_1 = 0$. When G_1 is critical, according to its type of structure, we can obtain a corresponding reduced graph G_2 by deleting some vertices from G_1. s_2 is the search number of G_2 and t_2 indicates the structure of G_2. Continue this procedure until the reduced graph is non-critical or empty.

Type 1 $s(G[e_1]) = s(G[e_2]) = s(G_i)$. The reduced graph G_{i+1} is obtained by deleting all the vertices of $G[v_i]$ except v_i from G_i.

Type 2 $s(G[e_1]) = s(G[e_2]) = s(G_i)$. The reduced graph G_{i+1} is obtained by deleting all the vertices of $G[C_i]$ except z from G_i.

Type 3 $s(X \cup Y \cup \{x_2 y_1\}) = s(G[e_1]) = s(G_i)$. The reduced graph G_{i+1} is obtained by deleting all the vertices of $G[C_i]$ except z from G_i.

Type 4 $s(G[e_1]) = s(G[e_2]) = s(G_i)$. The reduced graph G_{i+1} is obtained by deleting all the vertices of $G[x]$ except x from G_i.

Type 5 $s(X) = s(Y) = s(G_i)$. The reduced graph G_{i+1} is obtained by deleting all the vertices of $G[x]$ except x and all the vertices of $G[y]$ except y from G_i.

The precise definition of a label is given as follows.

Definition 3.1 *Let $G[r]$ be a rooted 3CDG, the label of a vertex v (resp. non-cycle edge e or cycle C) in $G[r]$, denoted by $L(v)$ (resp. $L(e)$ or $L(C)$), is defined as a sequence of elements $(s_1^{t_1}, s_2^{t_2}, \ldots s_m^{t_m})$. Each element $s_i^{t_i}$ consists of a positive integer s_i and a superscript t_i, where $0 \le t_i \le 5$. If $t_i = 0$, we call $s_i^{t_i}$ a non-critical element; otherwise, we call it a s_i-critical element of type-t_i. The value of $s_i^{t_i}$, denoted by $|s_i^{t_i}|$, is the positive integer s_i. The value of $L(v)$ (resp. $L(e)$ or $L(C)$), denoted by $|L(v)|$ (resp. $|L(e)|$ or $|L(C)|$), is the value of its first element s_1. $L(v)$ (resp. $L(e)$ or $L(C)$) satisfies the following conditions.*

1. *$s_1 > s_2 > \ldots > s_m$ and only the last element $s_m^{t_m}$ can be non-critical.*
2. *G_1 is $G[v]$ (resp. $G[e]$ or $G[C]$), and for $2 \le i \le m$, G_i is defined as a graph obtained from G_{i-1} according to t_{i-1}, see condition 3 for details.*
3. *If $m > 1$, then for $i = 1, 2, \ldots, m-1$, we have $s_i = s(G_i)$, $t_i > 0$ and*
   - *(a) if $t_i = 1$, there exists a non-cycle vertex v_i in G_i and v_i has two outgoing edges e_1 and e_2 such that $s(G[e_1]) = s(G[e_2]) = s_i$. G_{i+1} is defined as the graph obtained by deleting all the vertices of $G[v_i]$ except v_i from G_i.*
   - *(b) if $2 \le t_i \le 5$, there exists a cycle $C_i = zz_1x_1xx_2y_1yy_2z_2z$ (see Figure 1) in G_i. Let z be the entrance-vertex of C_i, let X be $G[x] \cup \{xx_1, xx_2\}$, Y be $G[y] \cup \{yy_1, yy_2\}$, and Z be $G[z] \cup \{zz_1, zz_2\}$. Assume $s(X) \ge s(Y)$, then we have*
     - *if $t_i = 2$, z has two outgoing edges e_1 and e_2 such that $s(G[e_1]) = s(G[e_2]) = s_i$. G_{i+1} is defined as the graph obtained by deleting all the vertices of $G[C_i]$ except z from G_i.*
     - *if $t_i = 3$, $s(X \cup Y \cup \{x_2y_1\}) = s_i$ and z has one outgoing edges e_1 such that $s(G[e_1]) = s_i$. G_{i+1} is defined as the graph obtained by deleting all the vertices of $G[C_i]$ except z from G_i.*
     - *if $t_i = 4$, x has two outgoing edges e_1 and e_2 such that $s(G[e_1]) = s(G[e_2]) = s_i$. G_{i+1} is defined as the graph obtained by deleting all the vertices of $G[x]$ except x from G_i.*
     - *if $t_i = 5$, $s(X) = s(Y) = s_i$. G_{i+1} is defined as the graph obtained by deleting all the vertices of $G[x]$ except x and all the vertices of $G[y]$ except y from G_i.*
4. *$s(G_m) = s_m$, $0 \le t_m \le 3$ and*
   - *(a) if $t_m = 1$, L must be the label of a vertex v, and v has two outgoing edges e_1 and e_2 such that $s(G[e_1]) = s(G[e_2]) = s_m$.*
   - *(b) if $2 \le t_m \le 3$, L must be the label of a cycle $C = zz_1x_1xx_2y_1yy_2z_2z$ (see Figure 1). Let z be the entrance-vertex of C, let X be $G[x] \cup \{xx_1, xx_2\}$, Y be $G[y] \cup \{yy_1, yy_2\}$, and Z be $G[z] \cup \{zz_1, zz_2\}$. Assume $s(X) \ge s(Y)$, then we have*
     - *if $t_i = 2$, z has two outgoing edges e_1 and e_2 such that $s(G[e_1]) = s(G[e_2]) = s_m$.*
     - *if $t_i = 3$, $s(X \cup Y \cup \{x_2y_1\}) = s_m$ and z has one outgoing edges e_1 such that $s(G[e_1]) = s_m$.*

For the first element $s_1^{t_1}$ of $L(v)$ (resp. $L(e)$ or $L(C)$), if $t_1 > 0$, $G[v]$ (resp. $G[e]$ or $G[C]$) is said to be s_1-*critical* of *type*-t_1, and the corresponding vertex v_1 or cycle C_1 is called the s_1-*critical vertex* or s_1-*critical cycle* in $G[v]$ (resp. $G[e]$ or $G[C]$).

Let S be a monotonic search strategy for a graph G and v be a vertex in G. During the procedure of performing S on G, if a searcher is placed on v and this searcher is never removed from v until G is cleared, then we say that S *ends at* v; if a searcher is placed on v in the first action of S and this searcher will never been removed from v until all the edges incident with v are cleared, then we say that S *starts from* v. If there is no optimal monotonic search strategy to clear G starting from or ending at v, then we call v a *bad vertex* of G.

For a graph G, let S be a search strategy of G that is represented by a sequence of actions, i.e., $S = (a_1, \ldots, a_k)$. The *reversal* of S, denoted by S^R, is defined as $S^R = (\overline{a}_k, \overline{a}_{k-1}, \ldots, \overline{a}_1)$, where each $\overline{a}_i$, $1 \le i \le k$, is the *converse* of a_i, which is defined as follows: the action "place a searcher on vertex v" and the action "remove a searcher from vertex v" are converse with each other; and the action "slide the searcher from v to u along the edge vu" and the action "slide the searcher from u to v along the edge uv" are converse with each other. It is easy to verify that if a search strategy S starts from a vertex v, then S^R ends at v. S^R also has the following property.

Lemma 3.2 *If S is an optimal monotonic search strategy of a graph G, then S^R is also an optimal monotonic search strategy of G.*

For two sequences of elements $A = (a_1, \ldots, a_k)$ and $B = (b_1, \ldots, b_m)$. The *concatenation* of A and B, denoted by $A \circ B$, is defined as $A \circ B = (a_1, \ldots, a_k, b_1, \ldots, b_m)$.

For the sake of simplicity, we will use a normalized 3CDG as the input of our algorithm. For each cycle C in a 3CDG, let x, y and z be the three vertices with degree more than 2. Note that there may not be three vertices each of which has degree more than 2 and in this case, we will choose the degree-two cycle vertex. Recall that there are at least 3 vertices in each cycle since we require that all the graphs in this paper are finite without loops and multiple edges. Replace each of $x \sim y$, $y \sim z$ and $z \sim x$ by a path of length three such that $C = zz_1x_1xx_2y_1yy_2z_2z$ (see Figure 2). This procedure takes $O(n)$ time. Notice that the search number of the normalized 3CDG equals the search number of the original graph.

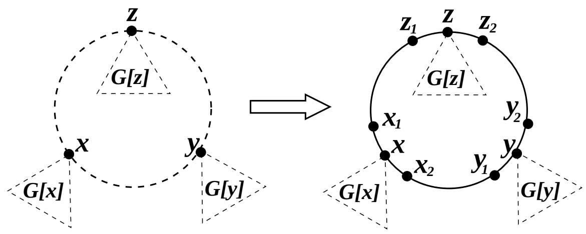

Figure 2: normalization of 3CDG

3.2 The main idea of the algorithm

The following algorithm SearchNumber-3CDG computes the labels of vertices, non-cycle edges and cycles in a rooted 3CDG $G[r]$ by the labeling method. And later we will construct the corresponding optimal search strategy based on these labels.

Algorithm SearchNumber-3CDG($G[r]$)
Input: A rooted 3CDG $G[r]$.
Output: Labels of all vertices, cycles and non-cycle edges.

1. Assign label (0^0) to each leaf (except r if r is also a leaf), (1^0) to each pendant edge and (2^0) to each pendant cycle in $G[r]$.
2. If r is labeled, then return labels of all vertices, cycles and non-cycle edges in $G[r]$.
3. - For each vertex v whose all out-going edges have been labeled, compute the label $L(v)$.
   - For each cycle C in which all the vertices with degree more than two have been labeled, compute the label $L(C)$.
   - For each non-cycle edge (u, v), if v is on a labeled cycle or if v is a labeled non-cycle vertex, then compute the label $L(uv)$.
   - Go to Step 2.

It is easy to verify that the label of a pendant edge is (1^0) and the label of a pendant cycle is (2^0). Next we will introduce: (i) how to compute the label of a vertex if all its out-going edges have been labeled; (ii) how to compute the label of a cycle if all the three cycle vertices with degree more than 2 have been labeled; and (iii) how to compute the label of a non-cycle edge if its head is on a labeled cycle or its head is a labeled non-cycle vertex.

3.3 Computing the label of a vertex

Lemma 3.3 [2] *Let H be a subgraph of G with $s(H) = k$. When we perform an optimal search strategy on G, there must exist a vertex h in H such that there are at least k searchers in H after placing a searcher on h.*

Lemma 3.4 *Let G be a graph containing three connected subgraphs G_1, G_2 and G_3, whose vertex sets are pairwise disjoint, such that for every pair G_i and G_j there exists a path in G between G_i and G_j that contains no vertex in the third subgraph. If $s(G_1) = s(G_2) = s(G_3) = k$, then $s(G) \geq k + 1$.*

Proof. Assume that $s(G) = k$ and let S be an arbitrary monotonic search strategy to clear G using k searchers. Let h_i, $1 \leq i \leq 3$, be the vertex in G_i described in Lemma 3.3. W.l.o.g, let h_1, h_2, h_3 be the order in which S places searchers on them. Since G_1, G_2 and G_3 are pairwise vertex-disjoint and there is a path between h_1 and h_3 containing no vertex of G_2. At the moment after a searcher is placed on h_2, all k searchers are in G_2 and there is no searcher can be used to protect the path between h_1 and h_3 from recontamination, which contradicts our initial assumption. ■

We now consider how to compute the label of a vertex v when the labels of all its outgoing edges are known. Suppose v has d children, $v_1, \ldots, v_d$. For $1 \le i \le d$, let $L(vv_i)$ be the label of (v, v_i) that contains m_i elements. For $1 \le j \le m_i$, $s_{j,i}^{t_{j,i}}$ is the j-th element in $L(vv_i)$. Here we use additional subscripts in s_j and t_j to indicate whose label it belongs to, i.e., $L(vv_i) = (s_{1,i}^{t_{1,i}}, s_{2,i}^{t_{2,i}}, \ldots s_{m_i,i}^{t_{m_i,i}})$. Then we have a graph $G_{m_i,i}$ that is defined in Definition 3.1, where $G[vv_i]$ and $G_{m_i,i}$ correspond to G_1 and G_{m_i} respectively. Let G_0 be the union of all $G_{m_i,i}$, for $1 \le i \le d$. Let L_0 be a multiset that contains all non-critical elements of each $L(vv_i)$, $1 \le i \le d$. Let p be the value of the largest elements in L_0 and q be the number of elements with value p in L_0. We first compute the label of v in $G_0[v]$, denoted by L_v. From Lemma 3.4, we have the following results.

(i) If $q \ge 3$, then $L_v = ((p+1)^0)$. We can clear G_0 by $p+1$ searchers ending at v using the following strategy. Place a searcher on v and use other p searchers to clear all the edge-branches of v one by one since each of them has search number at most p.

(ii) If $q = 2$, then $L_v = (p^1)$ and v is the p-critical vertex in $G_0[v]$. We can clear G_0 by p searchers using the following strategy. Let X and Y be the two edge-branches with search number p. First, clear X with p searchers ending at v. Then keep one searcher on v and use other $p-1$ searchers to clear all the other edge-branches of v except Y since each of them has search number at most $p-1$. Finally, clear Y with all p searchers starting from v.

(iii) If $q = 1$, then $L_v = (p^0)$. We can clear G_0 by p searchers ending at v using the following strategy. Let X be the edge-branch with search number p. First, clear X with p searchers ending at v. Then keep one searcher on v and use other $p-1$ searchers to clear all the other edge-branches of v since each of them has search number at most $p-1$.

After obtaining the label L_v of v in $G_0[v]$, we merge L_v with all critical elements of $L(vv_i)$, for $1 \le i \le d$, which can be done by function MergeLabel-Vertex. The output of this function is the label of v in $G[r]$.

Function MergeLabel-Vertex$(L_1, L_2, \ldots, L_d, L_v)$
Input: $L_1, L_2, \ldots, L_d$ and L_v, where L_i is the label of the i-th outgoing edge of v, for $1 \le i \le d$, and L_v is the label of v in $G_0[v]$ that contains only one element.
Output: The label of v in $G[r]$.

1. Set $\alpha \leftarrow$ the only element of L_v.
2. Let w be the value of the largest repeated critical elements in the labels $L_1, L_2, \ldots, L_d$.
   /* Two critical elements are repeated if they have the same value */
3. **if** $|\alpha| < w+1$, **then** $\alpha = (w+1)^0$.
4. Let L be a sequence containing all the critical elements of $L_1, L_2, \ldots, L_d$ with value larger than or equal to $|\alpha|$.
5. Set $h \leftarrow$ the value of the last element in L;
   if $h > |\alpha|$ **then return** $L(v) \circ (\alpha)$;
   else update L by deleting its last element;
   Set $\alpha \leftarrow (|\alpha|+1)^0$;
   Go to Step 5.

3.4 Computing the label of a cycle

Lemma 3.5 *Given a rooted 3CDG $G[r]$ that contains a cycle $C = zz_1x_1xx_2y_1yy_2z_2z$ (see Figure 2). $G[x]$, $G[y]$ and $G[z]$ are the vertex-branches of x, y and z respectively. Let X be $G[x] \cup \{xx_1, xx_2\}$, Y be $G[y] \cup \{yy_1, yy_2\}$, and Z be $G[z] \cup \{zz_1, zz_2\}$. If $s(X) = s(Y) = k$, $s(Z) \le k$ and neither $X[x]$ nor $Y[y]$ is k-critical, then $s(G) = k+1$ if and only if $s(Z) = k$.*

Proof. ($\Leftarrow$). If $s(X) = s(Y) = s(Z) = k$, it follows from Lemma 3.4 that $s(G) \ge k+1$. We now show that G can be cleared by $k+1$ searchers starting from z. We know that $s(G[z]) \le k$ since $G[z]$ is a subgraph of Z. We first station one searcher on z and use k searchers to clear $G[z]$; then slide one searcher from z along the path zz_2y_2y to y; slide the searcher stationed on z along the path zz_1x_1x to x; use one searcher to clear the path xx_2y_1y; use k searchers to clear $G[x]$ starting from x since $G[x]$ is not k-critical; finally, use k searchers to clear $G[y]$ starting from y since $G[y]$ is not k-critical either. Thus, G can be cleared by using $k+1$ searchers starting from z.

(⇒). Suppose $s(Z) < k$. We show that G can be cleared by k searchers. Let S be an optimal search strategy of Z. W.l.o.g., let us assume the edge zz_1 is cleared before zz_1 in S. First we clear $G[x]$ by k searchers ending at x; station one searcher on x and then perform S on Z. During this procedure, when we use one searcher to clear the edge zz_1, we clear the path zz_1x_1x instead and then slide the searcher on x along the path xx_2y_1y to y; when we use one searcher to clear the edge zz_2, we clear the path zz_2y_2y instead. After S is finished, X and Z and C are all cleared. Then we can use k searchers to clear $G[y]$ starting from y since $G[y]$ is not k-critical. ■

Lemma 3.6 *Given a rooted 3CDG $G[r]$ that contains a cycle $C = zz_1x_1xx_2y_1yy_2z_2z$ (see Figure 2). If $s(G[x]) = k$ and x has two outgoing edges e_1 and e_2 such that $s(G[e_1]) = s(G[e_2]) = k$ and none of these two edge-branches is k-critical. Let G^* be the graph obtained from G by deleting $G[x]$ (including x). Then $s(G) \geq k+1$ if and only if $s(G^*) \geq k$.*

Proof. (⇒). Suppose $s(G^*) < k$. We show that G can be cleared by k searchers. First, we use k searchers to clear $G[e_1]$ ending at x; keep one searcher on x and clear all the other edge-branches of x except $G[e_2]$ using other $k-1$ searchers since each of them has search number at most $k-1$; clear G^* and edges x_1x and x_2x using $k-1$ searchers since $s(G^*) < k$; finally, clear $G[e_2]$ starting from x using k searchers.

(⇐). If $s(G^*) \geq k$, it follows from Lemma 3.4 that $s(G) \geq k+1$. ■

We now consider how to compute the label of a cycle C when we know the labels of all the three cycle vertices with degree more than 2.

Let $C = zz_1x_1xx_2y_1yy_2z_2z$ be a cycle in a rooted 3CDG $G[r]$ and z be the entrance-vertex of C. Suppose $L(x)$, $L(y)$ and $L(z)$ are the labels of x, y and z respectively. Let X, Y and Z be the graphs defined as in Lemma 3.5. By using function MergeLabel-Vertex to merge $L(x)$ (resp. $L(y)$ or $L(z)$) with (1^1), we can obtain the label of x (resp. y or z) in $X[x]$ (resp. $Y[y]$ or $Z[z]$), denoted by L_x (resp. L_y or L_z). $L_x = (s_{1,x}^{t_{1,x}}, s_{2,x}^{t_{2,x}}, \ldots s_{m_x,x}^{t_{m_x,x}})$. Then we have a graph $G_{m_x,x}$ that is defined in Definition 3.1, where $X[x]$ and $G_{m_x,x}$ correspond to G_1 and G_{m_x} respectively. Similarly, we have $G_{m_y,y}$ and $G_{m_z,z}$. For simplicity, we use X', Y', Z' to denote $G_{m_x,x}, G_{m_y,y}, G_{m_z,z}$ respectively and a_X, a_Y, a_Z to denote $s_{m_x,x}^{t_{m_x,x}}, s_{m_y,y}^{t_{m_y,y}}, s_{m_z,z}^{t_{m_z,z}}$ respectively. Let p be the largest value among a_X, a_Y and a_Z. Let $G_0 = X' \cup Y' \cup Z' \cup C$. The following cases decide the label of the cycle C in $G_0[z]$, denoted by L_C.

Case 1: More than one of a_X, a_Y and a_Z are p-critical.
$L_C = ((p+1)^0)$. By Lemma 3.4, $s(G_0) \geq p+1$ and we can use $p+1$ searchers to clear G_0 ending at vertex z by following strategy. First, we station one searcher on vertex x; clear X' using p searchers since it has search number at most p. y has at most two edge-branches with search number p and none of them is p-critical. We clear one of the largest edge-branches of y using p searchers ending at y; use one searcher to clear the path xx_2y_1y; slide the searcher on x from x to z along the path xx_1z_1z and keep it on z; use one searcher to clear the path zz_2y_2y; clear all the other edge-branches of y except the second largest one using $p-1$ searchers since each of them has search number at most $p-1$; use p searchers to clear the second largest edge-branch of y starting from y; finally, clear Z' using p searchers since it has search number at most p.

Case 2: Only one of a_X, a_Y and a_Z is p-critical.

Case 2.1: a_Z is p-critical. Let $G^* = X' \cup Y' \cup \{x_2y_1\}$.

Case 2.1.1: $s(G^*) < p$.
$L_C = (p^2)$. By Lemma 3.6, $s(G_0) = p$ and we can clear G_0 using p searchers.

Case 2.1.2: $s(G^*) \geq p$.
$L_C = ((p+1)^0)$. By Lemma 3.6, $s(G_0) \geq p+1$. We can use $p+1$ searchers to clear G_0 ending at vertex z by a similar strategy described in case 1.

Case 2.2: a_X (or a_Y) is p-critical. Let $G^* = Z' \cup Y' \cup \{y_2z_2\}$ (or $G^* = Z' \cup X' \cup \{z_1x_1\}$).

Case 2.2.1: $s(G^*) < p$.
Let L be the label of z in $G^*[z]$. $L_C = (p^4) \circ L$. By Lemma 3.6, $s(G_0) = p$ and we can clear G_0 by p searchers.

Case 2.2.2: $s(G^*) \geq p$.
$L_C = ((p+1)^0)$. By Lemma 3.6, $s(G_0) \geq p+1$. We can clear G_0 by $p+1$ searchers ending at vertex z using a similar strategy described in case 1.

Case 3: None of a_X, a_Y and a_Z is p-critical.

CASE 3.1: $a_X = a_Y = a_Z = p$.
$L_C = ((p+1)^0)$. By Lemma 3.5, $s(G_0) \geq p+1$. We can clear G_0 by $p+1$ searchers ending at vertex z using a similar strategy described in CASE 1.

CASE 3.2: Exactly two of a_X, a_Y and a_Z have value p.

CASE 3.2.1: $a_X = a_Y = p$.
Let L be the label of z in $Z'[z]$. $L_C = (p^5) \circ L$. By Lemma 3.5, $s(G_0) = p$ and we can clear G_0 using p searchers.

CASE 3.2.2: $a_Z = p$ and ($a_X = p$ or $a_Y = p$).

CASE 3.2.2.1: One edge-branch of z in $Z'[z]$ has search number p.
$L_C = (p^3)$. By Lemma 3.6, $s(G_0) = p$ and we can clear G_0 using p searchers.

CASE 3.2.2.2: No edge-branch of z in $Z'[z]$ has search number p.
$L_C = (p^0)$. By Lemma 3.6, $s(G_0) = p$ and we can clear G_0 by p searchers ending at z by the following strategy. W.l.o.g, assume $|a_X| = p$. First, we use p searchers to clear X' ending at x. y has at most two edge-branches with search number $p-1$ and none of them is $(p-1)$-critical. We clear one of the largest edge-branches of y using $p-1$ searchers ending at y; use one searcher to clear the path xx_2y_1y; slide the searcher on x from x to z along the path xx_1z_1z and keep it on z; use one searcher to clear the path zz_2y_2y; clear all the other edge-branches of y except the second largest one using $p-2$ searchers since each of them has search number at most $p-2$; use $p-1$ searchers to clear the second largest edge-branch of y starting from y; finally, clear the edge-branches of z using $p-1$ searchers since each of them has search number at most $p-1$.

CASE 3.3: Only one of a_X, a_Y and a_Z has value p.

CASE 3.3.1: $a_Z = p$ and there is one edge-branch of z in $Z'[z]$ with search number p. Let $G^* = X' \cup Y' \cup \{x_2y_1\}$.

CASE 3.3.1.1: $s(G^*) = p$.
$L_C = (p^3)$. We can clear G_0 by p searchers using the following strategy. First, use p searchers to clear X' ending at x. y has at most two edge-branches with search number $p-1$ and none of them is $(p-1)$-critical. We clear one of the largest edge-branches of y using $p-1$ searchers ending at y; use one searcher to clear the path xx_2y_1y; silde the searcher on x from x to z along the path xx_1z_1z and keep it on z; use one searcher to clear the path zz_2y_2y; clear all the other edge-branches of y except the second largest one using $p-2$ searchers since each of them has search number at most $p-2$; use $p-1$ searchers to clear the second largest edge-branch of y starting from y; finally, we clear Z' using p searchers starting from z.

CASE 3.3.1.2: $s(G^*) < p$.
$L_C = (p^0)$. We can clear G_0 using p searchers ending at vertex z using the following strategy. First, use p searchers to clear the largest edge-branch of z ending at z and then keep one searcher on z; use $p-1$ searchers to clear all the other edge-branches of z since each of them has search number less than p; finally, use $p-1$ searchers to clear G^*.

CASE 3.3.2: $a_Z = p$ and no edge-branch of z in $Z'[z]$ has search number p.
$L_C = (p^0)$. We can use p searchers to clear G_0 ending at vertex z by a similar strategy described in CASE 3.2.2.2.

CASE 3.3.3: $a_X = p$ or $a_Y = p$.
$L_C = (p^0)$. We can use p searchers to clear G_0 ending at vertex z by a similar strategy described in CASE 3.2.2.2.

After obtaining the label L_C of C in $G_0[z]$, we will use the following function MERGELABEL-CYCLE to merge L_C with the three labels $L_x - a_X$, $L_y - a_Y$ and $L_z - a_Z$. The output of this function is the label of C in $G[r]$.

Function MERGELABEL-CYCLE(L_x, L_y, L_z, L_C)
Input: L_x, L_y, L_z and L_C, where L_x (resp. L_y or L_z) is the label of x (resp. y or z) in $X[x]$ (resp. $Y[y]$ or $Z[z]$) without the last element and L_C is the label of C in $G_0[z]$.
Output: The label of C in $G[r]$.

1. Set $\alpha \leftarrow$ the first element of L_C, $q \leftarrow |L_C|$.
2. Let w be the value of the largest repeated critical element of the input labels L_x, L_y and L_z. /* Two critical elements are repeated if they have the same value */
3. **if** $|\alpha| < w+1$, **then** $\alpha = (w+1)^0$.

4. Let L be a sequence containing all the critical elements of L_x, L_y and L_z with value larger than or equal to $|\alpha|$.
5. Set $h \leftarrow$ the value of the last element in L;
   if $h > |\alpha|$ **then if** $|\alpha| = q$ **then return** $L(v) \circ L_C$;
   else return $L(v) \circ (\alpha)$;
   else update L by deleting its last element;
   Set $\alpha \leftarrow (|\alpha| + 1)^0$;
   Go to Step 5.

3.5 Computing the label of a non-cycle edge

Let $G[r]$ be a rooted 3CDG and (u, v) be a non-cycle edge in $G[r]$. If v is on a labeled cycle or v is a labeled non-cycle vertex, then function EdgeLabel computes the label of (u, v) in $G[r]$, denoted by $L(uv)$.

Function EdgeLabel($G[uv]$)
Input: The label of v, denoted by L.
Output: The label of the edge (u, v).

1. Let p be the last element of L.
2. **if** p is not critical, **then return** L.
3. **if** p is critical with value larger than 1, **then return** $L \circ (1^0)$.
4. **if** p is critical with value 1,
   then q is the smallest positive integer such that no element in L has value q;
   Update L by deleting all the elements with value less than q;
   return $L \circ (q^0)$.

3.6 Correctness and time complexity

Lemma 3.7 *Function* MergeLabel-Vertex *outputs the label of a vertex in* $G[r]$.

Proof. Let $G[r]$ be a rooted 3CDG and v be a vertex in $G[r]$. Let $v_1, \ldots, v_d$ be the d children of v. For $1 \le i \le d$, let $L_i = (s_{1,i}^{t_{1,i}}, s_{2,i}^{t_{2,i}}, \ldots s_{m_i,i}^{t_{m_i,i}})$ be the label of (v, v_i) in $G[r]$. From Definition 3.1, the last element of each L_i is non-critical. Let L_v be the label of v in $G_0[v]$, which is defined in Section 3.3. The input of function MergeLabel-Vertex is $L_1, L_2, \ldots, L_d$ and L_v.

Line 1-2: Recall that α is the only element of L_v and w is the value of the largest repeated critical elements among all elements in $L_1, L_2, \ldots, L_d$.

Line 3: There two cases regarding the value of $|\alpha|$.

1. $|\alpha| < w + 1$. For each i, $1 \le i \le d$, we can find the index j such that in L_i, $|s_{j-1,i}^{t_{j-1,i}}| > w \ge |s_{j,i}^{t_{j,i}}|$ (if $w \ge |s_{1,i}^{t_{1,i}}|$, then $j = 1$). Then we have a graph $G_{j,i}$ that is defined in Definition 3.1, where $G[vv_i]$ and $G_{j,i}$ correspond to G_1 and G_j respectively. Let G' be the union of all $G_{j,i}$, $1 \le i \le d$. Since w is the value of the largest repeated critical elements, there are at least two disjoint w-critical branches in G'. By Lemma 3.4, $s(G') \ge w + 1$. And G' can be cleared by $w + 1$ searchers starting from v: station one searcher on v and use other w searchers to clear every edge-branch of v since it has search number at most w. Thus $s(G') = w + 1$ and update $\alpha = (w + 1)^0$.

2. $|\alpha| \ge w + 1$. For each i, $1 \le i \le d$, we can find the index j such that in L_i, $|s_{j-1,i}^{t_{j-1,i}}| \ge |\alpha| > |s_{j,i}^{t_{j,i}}|$ (if $|s_{m_i,i}^{t_{m_i,i}}| \ge |\alpha|$, then $j = m_i$). Then we have a graph $G_{j,i}$ that is defined in Definition 3.1, where $G[vv_i]$ and $G_{j,i}$ correspond to G_1 and G_j respectively. It is easy to see that each $G_{j,i}$ has search number less than or equal to $|\alpha|$ and if its search number equals $|\alpha|$, then it is not $|\alpha|$-critical. Let G' be the union of all $G_{j,i}$, $1 \le i \le d$. First, we know $s(G') \ge |\alpha|$ since G' contains G_0 as its subgraph and $s(G_0) = |\alpha|$. Then, notice that at most two of $G_{j,i}$ have search number equal to $|\alpha|$ and G' can be cleared by $|\alpha|$ searchers. So $s(G') = |\alpha|$.

After Line 3, we have $s(G') = |\alpha|$ and v has no $|\alpha|$-critical edge-branch in $G'[v]$ (v may have one $|\alpha|$-critical vertex-branch and in such case, v must be the $|\alpha|$-critical vertex).

Line 4: Let L be the sequence formed by all the critical elements of the input labels $L_1, L_2, \ldots, L_d$ with value larger than or equal to $|\alpha|$. Notice that L contains no repeated element.

Line 5: For each i, $1 \le i \le d$, we can find the index j such that in L_i, $|s_{j-1,i}^{t_{j-1,i}}| > |\alpha| \ge |s_{j,i}^{t_{j,i}}|$ (if $|\alpha| \ge |s_{1,i}^{t_{1,i}}|$, then $j = 1$). Then we have a graph $G_{j,i}$ that is defined in Definition 3.1, where $G[vv_i]$ and $G_{j,i}$ correspond to G_1 and G_j respectively. Let G'' be the union of all $G_{j,i}$, $1 \le i \le d$. If the last element of L is $|\alpha|$-critical, which means a certain $G_{j,i}$ is $|\alpha|$-critical. We have $s(G') = |\alpha|$, where G' is defined in Line 3. G' and the $|\alpha|$-critical $G_{j,i}$ are edge disjoint. Thus, we have $s(G'') \ge |\alpha| + 1$. It is easy to clear G'' using $|\alpha| + 1$ searchers ending at v. Update $\alpha = (|\alpha| + 1)^0$ and update L by deleting the last element. Repeat this step until the last element of L has value larger than $|\alpha|$. At this time, $L \circ (\alpha)$ satisfies the definition of the label of v in $G[r]$. ■

Based on Lemma 3.7 and the discussion in Section 3.4, we have the following lemma.

Lemma 3.8 *Function* MergeLabel-Cycle *outputs the label of a cycle in* $G[r]$.

Lemma 3.9 *Function* EdgeLabel *outputs the label of a non-cycle edge in* $G[r]$.

Proof. Let $G[r]$ be a rooted 3CDG, and (u, v) be a non-cycle edge in $G[r]$ where v is on a labeled cycle or v is a labeled non-cycle vertex. Note that the label of (u, v) should be the same as the label of u if $G[uv]$ is the only one edge-branch at u. In that case, $G[u]$ has only one more edge (u, v) than $G[v]$ (resp. $G[C]$ if v is on a labeled cycle C). The work done by function EdgeLabel is to merge the label of v (resp. C) with a label (1^0) that is the label of a single edge. Thus, by Lemma 3.7, function EdgeLabel outputs the label of (u, v) in $G[r]$. ■

From Lemmas 3.7, 3.8 and 3.9, SearchNumber-3CDG can compute the labels of each vertex, cycle and non-cycle edge. In the rest of this section we will analyze the time complexity of this algorithm.

We introduce a data structure used in [6] that compresses the label representation. For a sub-list of value consecutive critical elements in a label, we use an interval to represent them. For example, the label $(9^{t_1}, 8^{t_2}, 7^{t_3}, 6^{t_4}, 5^{t_5}, 3^{t_6}, 2^{t_7}, 1^0)$ is represented as $((9, 5), (3, 2), 1^0)$. Note that the non-critical element is not put into any interval. The benefit of this representation is to improve the label merging operation. For example, if we want to merge label $(9^{t_1}, 8^{t_2}, 7^{t_3}, 6^{t_4}, 5^{t_5}, 3^{t_6}, 2^{t_7}, 1^0)$ with (5^0), we can obtain the result (10^0) in one step by using this compressed representation.

Lemma 3.10 *The time complexity of function* EdgeLabel *is* $O(1)$ *with the compressed label representation.*

Proof. If the input label L is in the compressed label representation, we alter Lines 2, 3 and 4 of the function as follows.

2. **if** the last element of L is non-critical, **then return** L.
3. **if** the last element of L is (x, y) and $x \geq y > 1$, **then return** $L \circ (1^0)$.
4. **if** the last element of L is $(x, 1)$ and $x \geq 1$, **then return** $(L - (x, 1)) \circ ((x + 1)^0)$.

It is easy to verify that these operations are equivalent to the original ones and each operation takes constant time. ■

Lemma 3.11 *The time complexity of function* MergeLabel-Vertex *is* $O(|L_2|+d)$*, where* L_2 *is the second largest label among* $L_1, L_2, \ldots, L_d$.

Proof. Lines 1 and 3 take constant time. Line 5 also takes $O(1)$ time by using the same technique as in function EdgeLabel. We now consider the time complexity of Lines 2 and 4.

For $1 \leq i \leq d$, suppose that $|L_1| \geq |L_2| \geq |L_i|$ for $3 \leq i \leq d$. In order to achieve $O(|L_2| + d)$ time for MergeLabel-Vertex, we first merge part of L_1 with all the other labels and then merge the result with the rest of L_1. Replace Line 2 by the following fragment.

```
2.1 Set w ← 0, and remove elements with value less than or equal to |L_2| from L_1 and put them into Y.
2.2 for i = 2 to d, do
        for j = 1 to m_i, do
        /* L_i contains m_i critical elements and let s_j be the j-th largest one in L_i. */
            if no element in Y has value |s_j|,
              then put s_j into Y;
              else if |s_j| > w, then w = |s_j|;
                   break the inner for loop;
2.3 Delete all elements with value less than or equal to w from Y.
2.4 Represent Y by the compressed form.
```

Line 2.1 takes $O(|L_2|)$ time. In Line 2.2, each time when we check an element, we either add it into Y or finish checking the label that contains it. The size of Y is at most $|L_2|$ and there are d labels. Thus, Line 2.2 takes $O(|L_2| + d)$ time and Lines 2.3 and 2.4 take $O(|L_2|)$ time. Hence, the time complexity of Line 2 is $O(|L_2| + d)$.

After the execution of this fragment, w is the value of the largest repeated critical elements in $L_1, L_2, \ldots, L_d$ and Y contains all the critical elements with value less than or equal to $|L_2|$ and larger than w (if there is no such element, Y is empty). Consider the following two cases regarding the value of α after Line 3.

Case 1. $|\alpha| > |L_2| + 1$. Then critical elements with value larger than or equal to $|\alpha|$ can only appear in L_1. Let L be L_1 and Line 4 takes $O(1)$ time.

CASE 2: $|\alpha| \leq |L_2| + 1$. Let $L = L_1 \circ Y$ and delete all elements in L with value smaller than $|\alpha|$. Line 4 takes $O(|L_2|)$ time.

Therefore, the time complexity of function MERGELABEL-VERTEX is $O(|L_2| + d)$. ∎

Similarly to Lemma 3.11, we have the following lemma.

Lemma 3.12 *The time complexity of function* MERGELABEL-CYCLE *is $O(\beta)$, where β is the value of the second largest label among L_x, L_y and L_z.*

Lemma 3.13 *If a function $f(n)$ is defined on the positive integers by the recurrence equation*

$$f(n) = \begin{cases} c, & n = 1, 2, \\ f(m_1) + c, & k = 1, n \geq 3, \\ \max_M\{\sum_{i=1}^{k} f(m_i) + c(\lceil \log m_2 \rceil + k)\}, & k \geq 2, n \geq 3, \end{cases}$$

where $M = \{(m_1, m_2, \ldots, m_k) : m_1 \geq m_2 \geq \cdots m_k \geq 1$, and $\sum_{i=1}^{k} m_i = n - 2\}$, and $c \geq 1$ is a constant, then $f(n)$ is $O(n)$.

PROOF. We use induction to show that $f(n) \leq c(3n - \lceil \log n \rceil - 1)$. It is easy to verify that this is true for $n = 1, 2$. Assume the inequality is true for any $n \leq N - 1$. Now let us consider $n = N$.

Case 1: $k = 1$.

$$\begin{aligned} f(N) &= f(m_1) + c \\ &= f(N-1) + c \\ &\leq c(3(N-1) - \lceil \log(N-1) \rceil - 1) + c \\ &= c(3N - (2 + \lceil \log(N-1) \rceil) - 1) \\ &\leq c(3N - \lceil \log N \rceil - 1). \end{aligned}$$

Case 2: $k \geq 2$.

Let $k_1 = |\{m_i | m_i \geq 2, 1 \leq i \leq k\}|$ and $k_2 = |\{m_i | m_i = 1, 1 \leq i \leq k\}|$. We have that $k_1 + k_2 = k$ and $\sum_{i=k_1+1}^{k} f(m_i) = c \cdot k_2$.

$$\begin{aligned} f(N) &= \max\{\sum_{i=1}^{k} f(m_i) + c(\lceil \log m_2 \rceil + k)\} \\ &= \max\{\sum_{i=1}^{k_1} f(m_i) + \sum_{i=k_1+1}^{k} f(m_i) + c(\lceil \log m_2 \rceil + k)\} \\ &= \max\{\sum_{i=1}^{k_1} f(m_i) + c(k_2 + \lceil \log m_2 \rceil + k)\} \\ &\leq \max\{c(3\sum_{i=1}^{k_1} m_i - \sum_{i=1}^{k_1} \lceil \log m_i \rceil - k_1) + c(k_2 + \lceil \log m_2 \rceil + k)\} \\ &= \max\{c(3(N - 1 - k_2) - k_1 + k_2 + k - \lceil \log m_1 \rceil - \sum_{i=3}^{k_1} \lceil \log m_i \rceil)\} \\ &= \max\{c(3N - (2 + k_2 + \sum_{i=3}^{k_1} \lceil \log m_i \rceil + \lceil \log m_1 \rceil) - 1)\} \\ &\quad (\text{note that } \sum_{i=3}^{k} \lceil \log m_i \rceil = 0 \text{ when } k = 2) \\ &\leq c(3N - \lceil \log N \rceil - 1). \end{aligned}$$

Let $\Delta = 2 + k_2 + \sum_{i=3}^{k_1} \lceil \log m_i \rceil + \lceil \log m_1 \rceil$. Now we show that $\Delta \geq \lceil \log N \rceil$. Case 2.1: $k_1 = 0$. We have that $k = k_2 = N - 1$ and $m_1 = 1$. In this case, $\Delta = 1 + N \geq \lceil \log N \rceil$. Case 2.2: $1 \leq k_1 \leq 2$. We have that $k = k_1 + k_2 \leq 2 + k_2$. In this case, $\Delta = 2 + k_2 + \lceil \log m_1 \rceil \geq k + \lceil \log m_1 \rceil \geq \lceil \log N \rceil$. Case 2.3: $k_1 \geq 3$. For $3 \leq i \leq k_1$, we have that $\lceil \log m_i \rceil \geq 1$ since $m_i \geq 2$. In this case, $\Delta = 2 + k_2 + \sum_{i=3}^{k_1} \lceil \log m_i \rceil + \lceil \log m_1 \rceil \geq 2 + k_2 + k_1 - 2 + \lceil \log m_1 \rceil = k + \lceil \log m_1 \rceil \geq \lceil \log N \rceil$. ∎

In algorithm SearchNumber-3CDG, we use $f(G[v])$ to denote the time used to compute the label of vertex v and use $f(G[C])$ to denote the time used to compute the label of cycle C. Then we have

$$f(G[v]) = f(G[vv_1]) + f(G[vv_2]) + \cdots + f(G[vv_d]) + O(s(G[vv_2]) + d),$$

where v has d edge-branches and $G[vv_2]$ is the second largest edge-branch according to their search numbers.

$$f(G[C]) = f(G[x]) + f(G[y]) + f(G[z]) + O(s(G^*)),$$

where x, y and z are the three vertices on C with degree more than 2, and G^* is one of $G[x]$, $G[y]$ and $G[z]$ that has the second largest search number.

The search number of a tree is $O(\log n)$, where n is the number of vertices in that tree. And from Theorem 5.1, the search number of a 3CDG is also $O(\log n)$. Theorem 3.14 follows from Lemma 3.13.

Theorem 3.14 *For a rooted 3CDG $G[r]$, algorithm* SearchNumber-3CDG *computes the labels of all vertices, cycles and non-cycle edges in O(n) time.*

3.7 Constructing an optimal search strategy

Theorem 3.15 *Let $G[r]$ be a rooted 3CDG. If the labels of all vertices, cycles and non-cycle edges are known, then we can construct an optimal search strategy for G in O(n) time.*

Proof. The following algorithm Search3CDG constructs an optimal search strategy of a rooted 3CDG.

Algorithm Search3CDG($G[r]$)
Input: A rooted 3CDG $G[r]$ and the labels of all vertices, cycles and non-cycle edges. The label of root r, $L(r) = (s_1^{t_1}, s_2^{t_2}, \dots s_m^{t_m})$.
Output: An optimal search strategy of $G[r]$.

if $G[r]$ is a single edge (r, v), **then return** ("place a searcher on v",
"slide the searcher from v to r along edge vr");
if $G[r]$ is a single cycle, **then return** ("place a searcher on r", "use another searcher to slide along the cycle starting from r and ending at r");

Case 1: $m = 1$, $L(r)$ contains only one element $s_1^{t_1}$;

Case 1.1: $t_1 = 0$

Case 1.1.1: r is a non-cycle vertex;
if r has only one child v
then S =Search3CDG($G[v]$);
return $S\circ$("slide the searcher on v to r along edge vr");
else S_i =Search3CDG($G[rv_i]$) for $1 \le i \le d$;
/* $v_1, \dots, v_d$ are children of r and $G[rv_1]$ is the edge-branch with the largest searcher number. */
return $S_1 \circ S_2 \circ \cdots \circ S_d$;

Case 1.1.2: r is the entrance-vertex of a cycle C;
/* Let x and y be the other two vertices on C with degree more than 2, assume $s(G[x]) \ge s(G[y])$. */
if $s(G[r]) = s_1$
then S_1 =Search3CDG($G[r]$);
S_2 =Search3CDG(G^*);
/* G^* is defined in Lemma 3.6. */
return $S_1 \circ S_2$;
else S_i =Search3CDG($G[yy_i]$) for $1 \le i \le d$;
/* $y_1, \dots, y_d$ are d children of y, $G[yy_1]$ and $G[yy_2]$ are the two edge-branches with the largest searcher number. */
S_p =Search3CDG($G[x]$);
S_q =Search3CDG($G[r]$);
$S_w \leftarrow$ ("clear the path xx_2y_1y", "slide the searcher on x to r along the path xx_1r_1r", "clear the path rr_2y_2y");
return $S_p \circ S_1 \circ S_3 \circ \cdots \circ S_d \circ S_w \circ S_2^R \circ S_q$;

Case 1.2: $t_1 = 1$

S_i =Search3CDG($G[rv_i]$) for $1 \le i \le d$;
/* $v_1, \dots, v_d$ are children of r and $G[rv_1]$ and $G[rv_2]$ are two edge-branches with the largest searcher number. */
return $S_1 \circ S_3 \circ \cdots \circ S_d \circ S_2^R$;

Case 1.3: $t_1 = 2$

S_i =Search3CDG($G[rv_i]$) for $1 \le i \le d$;
/* $v_1, \dots, v_d$ are children of r and $G[rv_1]$ and $G[rv_2]$ are two edge-branches with the largest searcher number. */

S_p =SEARCH3CDG(G^*);
/* G^* is defined in Lemma 3.6. */
return $S_1 \circ S_3 \circ \cdots \circ S_d \circ S_p \circ S_2^R$;

Case 1.4: $t_1 = 3$

S_i =SEARCH3CDG($G[yy_i]$) for $1 \le i \le d$;
/* $y_1, \ldots, y_d$ are children of y and $G[yy_1]$ and $G[yy_2]$ are two edge-branches with the largest searcher number. */
S_p =SEARCH3CDG($G[x]$);
S_q =SEARCH3CDG($G[r]$);
$S_w \leftarrow$ ("clear the path xx_2y_1y", "slide the searcher on x to z along the path xx_1z_1z", "clear the path zz_2y_2y");
return $S_p \circ S_1 \circ S_3 \circ \cdots \circ S_d \circ S_w \circ S_2^R \circ S_q^R$;

Case 2: $m > 1$, $L(r)$ contains more than one element;
/* Let $G[r]$ (or $G[C]$ if r is on cycle C) be G_1, and G_2 be defined in Definition 3.1.
Let x, y and z be the three vertices on C_1 with degree more than 2 and z be the entrance-vertex of C_1. */

Case 2.1: $t_1 = 1$

S_i =SEARCH3CDG($G[v_1u_i]$) for $1 \le i \le d$;
/* $u_1, \ldots, u_d$ are children of v_1 and $G[v_1u_1]$ and $G[v_1u_2]$ are two edge-branches with searcher number s_1. */
S_p =SEARCH3CDG(G_2);
return $S_1 \circ S_3 \circ \cdots \circ S_d \circ S_p \circ S_2^R$;

Case 2.2: $t_1 = 2$

S_i =SEARCH3CDG($G[zz_i]$) for $1 \le i \le d$;
/* $z_1, \ldots, z_d$ are children of z and $G[zz_1]$ and $G[zz_2]$ are two edge-branches with searcher number s_1. */
S_p =SEARCH3CDG(G_2);
S_q =SEARCH3CDG(G^*);
/* G^* is defined in Lemma 3.6. */
return $S_1 \circ S_3 \circ \cdots \circ S_d \circ S_p \circ S_q \circ S_2^R$;

Case 2.3: $t_1 = 3$

S_i =SEARCH3CDG($G[yy_i]$) for $1 \le i \le d$;
/* $y_1, \ldots, y_d$ are children of y and $G[yy_1]$ and $G[yy_2]$ are two edge-branches with the largest searcher number. */
S_p =SEARCH3CDG($G[x]$);
S_q =SEARCH3CDG($G[z]$);
S_h =SEARCH3CDG(G_2);
$S_w \leftarrow$ ("clear the path xx_2y_1y", "slide the searcher on x to z along the path xx_1z_1z", "clear the path zz_2y_2y");
return $S_p \circ S_1 \circ S_3 \circ \cdots \circ S_d \circ S_w \circ S_2^R \circ S_h \circ S_q^R$;

Case 2.4: $t_1 = 4$

S_i =SEARCH3CDG($G[xx_i]$) for $1 \le i \le d$;
/* $x_1, \ldots, x_d$ are children of x and $G[xx_1]$ and $G[xx_2]$ are two edge-branches with searcher number s_1. */
S_p =SEARCH3CDG(G_2);
return $S_1 \circ S_3 \circ \cdots \circ S_d \circ S_p \circ S_2^R$;

Case 2.5: $t_1 = 5$

S_1 =SEARCH3CDG($G[x]$);
S_2 =SEARCH3CDG($G[y]$);
S_3 =SEARCH3CDG(G_2);
Modify S_3 by inserting the action "slide the searcher on x to y along path xx_2y_1y" after the action by which edge x_1x is cleared;
return $S_1 \circ S_3 \circ S_2^R$;

The correctness of this algorithm follows from our discussion in Sections 3.3, 3.4 and 3.5. Notice that if $L(r)$ contains only one element and this element is non-critical, algorithm SEARCH3CDG will return an optimal search strategy of $G[r]$ ending at r.

We now analyze the running time of this algorithm. The work done by this algorithm outside the recursive calls take $O(1)$ time. Each time when the algorithm invokes itself, the input 3CDG will be divided into several non-empty edge disjoint subgraphs. Since a 3CDG has $O(n)$ edges, where n is the number of vertices, this algorithm invokes itself for at most $O(n)$ times. Therefore, the total running time of SEARCH3CDG is $O(n)$. ■

4 k-ary cycle-disjoint graphs

A *complete k-ary tree T* is a rooted k-ary tree in which all leaves have the same depth and every internal vertex has k children. If we replace each vertex of T with a $(k+1)$-cycle such that each vertex of internal cycle has degree at most 3, then we obtain a cycle-disjoint graph G, which we call a *k-ary cycle-disjoint graph (k-ary CDG)*. In T, we define the level of the root be 1 and the level of a leaf be the number of vertices in the path from the root to that leaf. We use T_k^h to denote a complete k-ary tree with level h and G_k^h to denote the k-ary CDG obtained from T_k^h (see Figure 3).

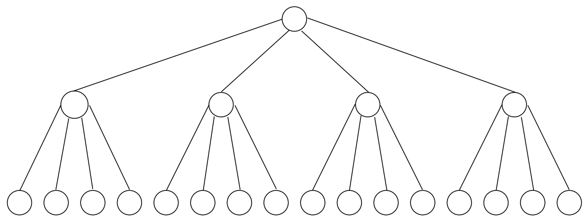

Figure 3: G_4^3, a 4-ary CDG with level 3

In this section, we will show how to compute the search numbers of k-ary CDGs. Similar to [7], we have the following lemmas.

Lemma 4.1 *For a connected graph G, let $C = v_1v_2\ldots v_mv_1$ be a cycle in G such that each v_i $(1 \le i \le m)$ connects to a connected subgraph X_i by a bridge. If $s(X_i) \le k$, $1 \le i \le m$, then $s(G) \le k+2$.*

Lemma 4.2 *For a connected graph G, let v_1, v_2, v_3, v_4 and v_5 be five vertices on a cycle C in G such that each v_i $(1 \le i \le 5)$ connects to a connected subgraph X_i by a bridge. If $s(X_i) \ge k, 1 \le i \le 5$, then $s(G) \ge k+2$.*

Lemma 4.3 *For a connected graph G, let $C = v_1v_2v_3v_4v_1$ be a 4-cycle in G such that each v_i $(1 \le i \le 4)$ connects to a connected subgraph X_i by a bridge. If $s(G) = k+1$ and $s(X_i) = k, 1 \le i \le 4$, then for any optimal search strategy of G, the first cleared vertex and the last cleared vertex must be in two distinct graphs $X_i + v_i$, $1 \le i \le 4$.*

Lemma 4.4 *For a CDG G with search number k, let S be an optimal monotonic search strategy of G in which the first cleared vertex is a and the last cleared vertex is b. If there are two cut-vertices a' and b' in G such that an edge-branch $G_{a'}$ of a' contains a and an edge-branch $G_{b'}$ of b' contains b and the graph G' obtained by removing $G_{a'}$ and $G_{b'}$ from G is connected, then we can use k searchers to clear G' starting from a' and ending at b'.*

Lemma 4.5 *Let G be a connected graph and C be a cycle of length at least four in G, and v_1 and v_2 be two vertices of C such that each v_i $(1 \le i \le 2)$ connects to a connected subgraph X_i by a bridge v_iv_i'. If $s(X_1) = s(X_2) = k$ and v_1' is a bad vertex of X_1 or v_2' is a bad vertex of X_2, then we need at least $k+2$ searchers to clear G starting from v_3 and ending at v_4, where v_3 and v_4 are any two vertices on C other than v_1 and v_2.*

Lemma 4.6 *For a connected graph G, let v_1, v_2, v_3 and v_4 be four vertices on a cycle C in G such that each v_i $(1 \le i \le 4)$ connects to a connected subgraph X_i by a bridge v_iv_i'. If $s(X_i) = k$, and v_i' is a bad vertex of X_i, $1 \le i \le 4$, then $s(G) \ge k+2$.*

From the above lemmas, we can prove the major result of this section.

Theorem 4.7 *Let T_k^h be a complete k-ary tree with level h and G_k^h be the corresponding k-ary CDG.*

(i) *If $k = 2$ and $h \ge 3$, then $s(T_2^h) = \lfloor \frac{h}{2} \rfloor + 1$ and $s(G_2^h) = \lfloor \frac{h}{2} \rfloor + 2$.*

(ii) *If $k = 3$ and $h \ge 2$, then $s(T_3^h) = h$ and $s(G_3^h) = h+1$.*

(iii) *If $k = 4$ and $h \ge 2$, then $s(T_4^h) = h$ and $s(G_4^h) = h + \lceil \frac{h}{2} \rceil$.*

(iv) *If $k \ge 5$ and $h \ge 2$, then $s(T_k^h) = h$ and $s(G_k^h) = 2h$.*

PROOF. The search numbers of complete k-ary trees can be verified directly by SEARCHNUMBER-3CDG since a tree can be regarded as a 3CDG without any cycle. Thus, we will only consider the search numbers of k-ary CDGs.

(i) The search number of G_2^h can be verified directly by SEARCHNUMBER-3CDG since G_2^h are 3CDGs.

(ii) We now prove $s(G_3^h) = h + 1$ by induction on h. Let $R = r_0r_1r_2r_3r_0$ be the cycle in G_3^h that corresponds to the root of T_3^h. Suppose r_0 is the vertex without any outgoing edges. When $h = 2$, it is easy to see that $s(G_3^2) = 3$ and all four vertices of R are not bad vertices in G_3^2. Suppose $s(G_3^h) = h + 1$ holds when $h < n$ and all four vertices of R are not bad vertices in G_3^h. When $h = n$, R has three edge-branches with search number n. It follows from Lemma 3.4 that $s(G_3^n) \geq n + 1$. We will show how to use $n + 1$ searchers to clear the graph by the following strategy: use n searchers to clear $G[r_1]$ ending at r_1; keep one searcher on r_1 and use n searchers to clear $G[r_2]$ ending at r_2; use one searcher to clear the edge r_1r_2; slide the searcher on r_1 to r_0 and slide the searcher on r_2 to r_3; use one searcher to clear the edge r_0r_3; then clear $G[r_3]$ with n searchers starting from r_3. This strategy never needs more than $n + 1$ searchers. Thus, $s(G_3^n) = n + 1$. From this strategy, it is easy to see that all four vertices of R are not bad vertices in G_3^n.

(iii) We will prove $s(G_4^h) = h + \lceil \frac{h}{2} \rceil$ by induction on h. Let $R = r_0r_1r_2r_3r_4r_0$ be the cycle in G_4^h that corresponds to the root of T_4^h. Suppose r_0 is the vertex without any outgoing edges. We want to show that if h is odd, then no bad vertex is on R, and if h is even, then r_0 is a bad vertex of G_4^h.

When $h = 2$, it is easy to see that $s(G_4^2) = 3$ and r_0 is a bad vertex in G_4^2. When $h = 3$, by Lemma 4.6, $s(G_4^3) \geq 5$ and it is easy to verify that 5 searchers can clear G_4^3 starting from any one of the five vertices on R. Suppose these results hold for G_4^h when $h < n$. We now consider the two cases when $h = n$.

If n is odd, $G[r_i]$ has search number $n - 1 + (n - 1)/2$ and r_i is a bad vertex in $G[r_i]$, $1 \leq i \leq 4$. By Lemma 4.6, we have $s(G_4^n) \geq n - 1 + (n - 1)/2 + 2 = n + (n + 1)/2$. We will show how to use $n + (n + 1)/2$ searchers to clear the graph by the following strategy. Let v be any one of the cycle vertex of R. We first place two searchers α and β on v and then slide β along R starting from v and ending at v. Each time when β arrives a vertex of R, we clear the subgraph attached to this vertex using $n - 1 + (n - 1)/2$ searchers. This strategy never needs more than $n + (n + 1)/2$ searchers. Thus, $s(G_4^n) = n + (n + 1)/2$. It is also easy to see that all five vertices of R are not bad vertices in G_4^n.

If n is even, $G[r_i]$ has search number $n - 1 + n/2$ and r_i is not a bad vertex in $G[r_i]$, $1 \leq i \leq 4$. By Lemma 3.4, we have $s(G_4^n) \geq n + n/2$. We will show how to use $n + n/2$ searchers to clear the graph by the following strategy: use $n - 1 + n/2$ searchers to clear $G[r_1]$ ending at r_1; use $n - 1 + n/2$ searchers to clear $G[r_2]$ ending at r_2; use one searcher to clear the edge r_1r_2; slide the searcher on r_1 along the path $r_1r_0r_4$ to r_4; slide the searcher on r_2 to r_3 along the edge r_2r_3; use one searcher to clear the edge r_3r_4; clear $G[r_3]$ with $n - 1 + n/2$ searchers starting from r_3 and finally clear $G[r_4]$ with $n - 1 + n/2$ searchers starting from r_4. This strategy never needs more than $n + n/2$ searchers. Thus, $s(G_4^n) = n + n/2$ and, by Lemma 4.3, r_0 is a bad vertex in G_4^n.

(iv) The search number of G_k^h, $k \geq 5$, can be verified directly from Lemmas 4.1 and 4.2. ■

5 Approximation algorithms

Megiddo et al. [12] introduced the concept of the *hub* and the *avenue* of a tree. Given a tree T with $s(T) = k$, only one of the following two cases must happen: (1) T has a vertex v such that all edge-branches of v have search number less than k, this vertex is called a *hub* of T; and (2) T has a unique path $v_1v_2 \ldots v_t$, $t > 1$, such that v_1 and v_t each has exactly one edge-branch with search number k and each v_i, $1 < i < t$, has exactly two edge-branches with search number k, this unique path is called an *avenue* of T.

Theorem 5.1 *Given a CDG G, if T is a tree obtained by contracting each cycle of G into a vertex, then $s(T) \leq s(G) \leq 2s(T)$.*

PROOF. Since T is a minor of G, we have $s(T) \leq s(G)$. We prove the second inequality by induction on $s(T)$. If $s(T) = 1$, T is a single vertex or a path, and correspondingly, G is a single vertex or a single cycle or a sequence of cycles connected by paths. It is easy to see that we can clear G using at most 2 searchers. Suppose $s(G) \leq 2s(T)$ holds for $s(T) \leq k - 1$, $k \geq 2$. When $s(T) = k$, we consider the following two cases.

1. T has a hub v. Every edge-branch of v in T has search number at most $k - 1$. By induction, the subgraph in G that corresponds to an edge-branch of v has search number at most $2(k - 1)$. If v corresponds to a vertex v' in G, then we can place a searcher on v' and use $2(k - 1)$ searchers to clear each subgraph attached to v'. If v corresponds to a cycle C in G, then let v' be a vertex of C. We first place two searchers α and β on v' and then slide β along C starting from v' and ending at v'. Each time when β arrives a vertex of C, we clear the subgraph attached to this vertex using $2(k - 1)$ searchers.

2. T has an avenue p. Every edge-branch attached to p has search number at most $k-1$. By induction, the subgraph in G that corresponds to an edge-branch attached to p has search number at most $2(k-1)$. The subgraph of G, say p', that corresponds to p is a path or a sequence of cycles connected by paths. In this case, we can use two searchers to clear p' while using $2(k-1)$ searchers to clear each subgraph attached to p'.

Hence, when $s(T) = k$, $s(G) \leq 2k$. Therefore, $s(T) \leq s(G) \leq 2s(T)$. For a k-ary CDG G_k^h and $k \geq 5$, by Theorem 4.7, $s(G_k^h) = 2s(T_k^h)$, which indicates that the second inequality is tight. ■

In Theorem 5.1, if G consists of two cycles linked by an edge, then T is an edge. Thus $s(T) = 1$ and $s(G) = 2$. Hence, the second inequality is tight.

Corollary 5.2 *For any CDG, there is an $O(n)$ time approximation algorithm with approximation ratio 2.*

Proof. Let G be a cycle-disjoint graph and S_G be the search strategy described in the proof of Theorem 5.1. Recall that S_G is constructed from S_T that is the optimal search strategy for T, where T is the tree obtained by contracting each cycle of G into a vertex. Since it takes linear time to compute S_T, it is easy to see that S_G can also be found in linear time. Let $\kappa(S_G)$ be the number of searchers required by S_G. We have

$$\frac{\kappa(S_G)}{s(G)} \leq \frac{2s(T)}{s(T)} = 2.$$

■

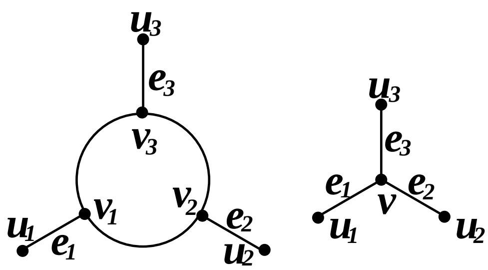

Figure 4: vertices of cycle and the corresponding tree nodes

Lemma 5.3 *Let T be the tree obtained from a 3CDG G by contracting every cycle of G into a vertex. If the degree of each cycle vertex in G is at most 3, then $s(G) \leq s(T) + 1$.*

Proof. Let $s(T) = k$ and let S_T be a monotonic search strategy to clear T using k searchers. We will show that G can be cleared using at most $k+1$ searchers by constructing a new search strategy S_G for G. S_G is a subsequence of S_T and contains some new actions that clear the cycle edges by an extra searcher λ.

Initially, let S_G be the same as S_T. Let C be a cycle of G and v be the corresponding vertex of T. In G, v_i and u_i are the two endpoints of e_i, $i = 1, 2, 3$. In T, v and u_i are the two endpoints of e_i, $i = 1, 2, 3$. See Figure 4. W.l.o.g., assume e_1, e_2 and e_3 of T are cleared in the described order. First, we modify some actions of S_G by the following operations: replace the action "place a searcher on v" by "place a searcher on v_1"; replace the action "slide a searcher along the edge $u_i v$ from u_i to v" by "slide a searcher along $u_i v_i$ from u_i to v_i", $i = 1, 2, 3$; replace the action "slide a searcher along the edge vu_i from v to u_i" by "slide a searcher along $v_i u_i$ from v_i to u_i", $i = 1, 2, 3$; replace the action "remove a searcher from v" by "remove a searcher from v_3".

Suppose e_2 is cleared in the i^{th} action of S_T. Then e_1 is cleared before the i^{th} action and there is at least one searcher α on v and α remains on v in the i^{th} action of S_T since the edge e_3 is still contaminated. There are two possible actions to clear e_2 in S_T.

Case 1 In S_T, a searcher β slides from v to u_2 by the i^{th} action. In this case, there is a corresponding β on v_1 in S_G at this moment. In this case, we insert five new actions immediately before the corresponding i^{th} action in S_G. "slide α from v_1 to v_3", "slide β from v_1 to v_2", "place λ on v_2", "slide λ from v_2 to v_3", "remove λ from v_3".

Case 2 In S_T, a searcher β slides from u_2 to v by the i^{th} action. In this case, there is a corresponding β on u_2 at this moment. In this case, we insert five new actions in S_G immediately after the corresponding i^{th} action in S_G. "place λ on v_2", "slide λ from v_2 to v_1", "remove λ from v_1", "slide α from v_1 to v_3", "slide β from v_2 to v_3".

Each time after we clear C by the new actions added in S_G, λ is free. It is easy to verify that S_G can clear G using no more than $k+1$ searchers. ■

Lemma 5.4 *Given a graph G, for any two vertices a and b of G, there is a search strategy that uses at most $s(G)+1$ searchers to clear G starting from a and ending at b.*

PROOF. Let S be an optimal search strategy of G. It follows from Lemma 3.2 that S^R is also an optimal search strategy of G. Vertex a is cleared before b either in S or in S^R. We first place a searcher λ on a and keep it on a; then perform the search strategy on G in which a is cleared before b; at the moment a is cleared, remove λ from a and place it on b and keep it on b until G is cleared. Thus, we can clear G starting from a and ending at b with no more than $s(G)+1$ searchers. ■

Definition 5.5 Let G be a connected graph and $X_1, X_2, \ldots, X_m$ be an edge partition of G such that each X_i is a connected subgraph and each pair of X_i share at most one vertex. Let G^* be a graph of m vertices such that each vertex of G^* corresponds to a X_i and there is an edge between two vertices of G^* if and only if the corresponding two X_i share a common vertex.

Theorem 5.6 *Let G, G^* and X_i, $1 \le i \le m$, be defined in Definition 5.5. If G^* is a path, then there is a search strategy that uses at most $\max_{1\le i\le m} s(X_i)+1$ searchers to clear G.*

PROOF. Suppose G^* is the path $v_1v_2\ldots v_m$ and v_i corresponds to X_i, $1 \le i \le m$. Let a_i be the vertex shared by X_i and X_{i+1}, $1 \le i \le m-1$ and let a_0 be a vertex in X_1 and a_m be a vertex in X_m. By Lemma 5.4, we can use $s(X_i)+1$ searchers to clear each X_i starting from a_{i-1} and ending at a_i, for $X_1, X_2, \ldots, X_m$. Therefore, there is a search strategy uses at most $\max_i s(X_i)+1$ searchers to clear G. ■

Theorem 5.7 *Let G, G^* and X_i, $1 \le i \le m$, be defined in Definition 5.5. If G^* is a tree, then there is a search strategy that uses at most $\max_{1\le i\le m} s(X_i) + \lceil\Delta(G^*)/2\rceil s(G^*)$ searchers to clear G, where $\Delta(G^*)$ is the maximum degree of G^*.*

PROOF. We prove the result by induction on $s(G^*)$. If $s(G^*)=1$, G^* is a single vertex or a path, $\lceil\Delta(G^*)/2\rceil = 1$, the result can be verified directly from Theorem 5.6. Suppose this result holds for $s(G^*) \le n$, $n \ge 2$. When $s(G^*) = n+1$, we consider the following two cases.

CASE 1. G^* has a hub v. Let $X(v)$ be the subgraph of G that corresponds to v and S be an optimal search strategy of $X(v)$. Each subgraph that corresponds to a neighbor of v in G^* shares a vertex with $X(v)$ in G. Divide these shared vertices into $\lceil\deg(v)/2\rceil$ pairs such that for each pair of vertices a_i and a_i', a_i is cleared before a_i' is cleared in S, $1 \le i \le \lceil\deg(v)/2\rceil$. Let v_i (resp. v_i') be the neighbor of v such that its corresponding subgraph of G, denoted by $X(v_i)$ (resp. $X(v_i')$), shares a_i (resp. a_i') with $X(v)$. Let v be the root of G^*, let T_i (resp. T_i') be the vertex-branch of v_i (resp. v_i') and let $X(T_i)$ (resp. $X(T_i')$) be the subgraph of G that is the union of the subgraphs that correspond to all vertices in T_i (resp. T_i'). Obviously a_i (resp. a_i') is the only vertex shared by $X(v)$ and $X(T_i)$ (resp. $X(T_i')$). Since v is a hub of G^*, we know that $s(T_i) \le n$. Thus, $s(X(T_i)) \le \max_i s(X_i) + \lceil\Delta(T_i)/2\rceil n \le \max_i s(X_i) + \lceil\Delta(G^*)/2\rceil n$. First, we place a searcher on each a_i, $1 \le i \le \lceil\deg(v)/2\rceil$. Then use $\max_i s(X_i) + \lceil\Delta(G^*)/2\rceil n$ searchers to clear each subgraph $X(T_i)$ separately. After that, we perform S to clear $X(v)$. Each time after some a_i is cleared by S, we remove the searcher on a_i and place it on a_i', $1 \le i \le \lceil\deg(v)/2\rceil$. Finally, after $X(v)$ is cleared, we again use $\max_i s(X_i) + \lceil\Delta(G^*)/2\rceil n$ searchers to clear each subgraph $X(T_i')$ separately. Therefore,we can clear G with no more than $\max_i s(X_i) + \lceil\Delta(G^*)/2\rceil n + \lceil\deg(v)/2\rceil \le \max_i s(X_i) + \lceil\Delta(G^*)/2\rceil(n+1)$ searchers.

CASE 2. G^* has an avenue $v_1v_2\ldots v_t$, $t>1$. Let v_0 be a neighbor of v_1 other than v_2 and let v_{t+1} be a neighbor of v_t other than v_{t-1}. Let $X(v_i)$, $0 \le i \le t+1$, be the subgraph of G that corresponds to v_i. For $0 \le i \le t$, let b_i be the vertex shared by $X(v_i)$ and $X(v_{i+1})$. For $1 \le i \le t$, let S_i be an optimal search strategy of $X(v_i)$ such that b_{i-1} is cleared before b_i is cleared. Thus, we can use a similar search strategy described in CASE 1 to clear each $X(v_i)$ and all the subgraphs that correspond to the edge-branches of v_i. Note that when we clear $X(v_i)$, b_{i-1} and b_i form a pair as a_i and a_i' in CASE 1. In such a strategy we never need more than $\max_i s(X_i) + \lceil\Delta(G^*)/2\rceil(n+1)$ searchers. ■

In Theorem 5.7, if each X_i is a unicyclic graph, then we have a linear time approximation algorithm for cycle-disjoint graphs. We can design a linear time approximation algorithm when each $s(X_i)$ can be found in linear time.

6 Conclusions

Our work mainly involves four aspects. First, we establish a linear time algorithm to compute the search number and the optimal search strategy for a 3-cycle-disjoint graph using the labeling method. Second, we improve the running time of the algorithm for computing the vertex separation and the optimal layout of a unicyclic graph from $O(n \log n)$ to $O(n)$. For a graph G, the search number of G equals the vertex separation of the 2-expansion of G. Thus, our improved algorithm can also compute the search number of a unicyclic graph. Third, we show how to compute the search number and the optimal search strategy of a k-ary cycle-disjoint graph. Finally, we prove several theorems that can be applied to design approximation algorithms for cycle-disjoint graphs and even more complicated graphs.

The results presented in the paper is a preliminary part of our research that will proceed to more complicated graphs with treewidth at most two. The roadmap we outlined was to find efficient algorithms to compute the search number of the following graphs one after another: trees, unicyclic graphs, CDGs and then outerplanar graphs. We have successfully found $O(n)$ algorithms for trees and unicyclic graphs and some classes of CDGs. In the future, finding efficient algorithms for computing the search number of graphs with constant treewidth continues to be a challenge.

References

[1] B. Alspach, Searching and searching graphs: A brief survey, Le Matematiche, 34pp.

[2] B. Alspach, X. Li, and B. Yang, Searching Graphs and Directed Graphs, *Manuscript*, 2004.

[3] H. Bodlaender, T. Kloks, Efficient and Constructive Algorithms for the Pathwidth and Treewidth of Graphs. *J. Algorithms* 21 (1996) 358–402.

[4] H. Bodlaender, F. Fomin, Approximation of pathwidth of outerplanar graphs. *J. Algorithms* 43 (2002) 190–200.

[5] D. Bienstock and P. Seymour, Monotonicity in graph searching, *Journal of Algorithms* 12 (1991) 239–245.

[6] J. Ellis, I. Sudborough and J. Turner, The vertex separation and search number of a graph, *Information and Computation* 113 (1994) 50–79.

[7] J. Ellis and M. Markov, Computing the vertex separation of unicyclic graphs, *Information and Computation* 192 (2004) 123–161.

[8] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

[9] N. Kinnersley, The vertex separation number of a graph equals its path-width, *Information Processing Letters* 42 (1992) 345–350.

[10] L. M. Kirousis and C. H. Papadimitriou, Searching and pebbling, *Theoretical Computer Science* 47 (1986) 205–218.

[11] A. S. LaPaugh, Recontamination does not help to search a graph. *Journal of ACM* 40 (1993) 224–245.

[12] N. Megiddo, S. L. Hakimi, M. Garey, D. Johnson and C. H. Papadimitriou, The complexity of searching a graph, *Journal of ACM* 35 (1988) 18–44.

[13] T. Parsons, Pursuit-evasion in a graph. *Theory and Applications of Graphs*, Lecture Notes in Mathematics, Springer-Verlag, pp. 426–441, 1976.

[14] S. Peng, C. Ho, T. Hsu, M. Ko, and C. Tang, Edge and Node Searching Problems on Trees, *Theoretical Computer Science* 240 (2000) 429–446.

[15] B. Yang, D. Dyer, and B. Alspach, Searching graphs with large clique number, 35 pp (submitted).