Searching Cycle-Disjoint Graphs

Boting Yang, Runtao Zhang, and Yi Cao

Department of Computer Science, University of Regina
{boting,zhang23r,caoyi200}@cs.uregina.ca

Abstract. In this paper, we study the problem of computing the minimum number of searchers who can capture an intruder hiding in a graph. We propose a linear time algorithm for computing the vertex separation and the optimal layout for a unicyclic graph. The best algorithm known so far is given by Ellis et al. (2004) and needs $O(n \log n)$ time, where n is the number of vertices in the graph. By a linear-time transformation, we can compute the search number and the optimal search strategy for a unicyclic graph in linear time. We show how to compute the search number for a k-ary cycle-disjoint graph. We also present some results on approximation algorithms.

1 Introduction

Given a graph in which an intruder is hiding on vertices or edges, the searching problem is to find the minimum number of searchers to capture the intruder. The graph searching problem has many applications [4,5,7].

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; and (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 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 move along a path that contains no searcher at a great speed at any time. For a graph G, the minimum number of searchers required to clear G is called the *search number*, denoted by $s(G)$. A search strategy for a graph G is *optimal* if this strategy can clear G using $s(G)$ searchers. Let $E(i)$ be the set of cleared edges just after action i. A search strategy is said to be *monotonic* if $E(i) \subseteq E(i+1)$ for every i. LaPaugh [8] and Bienstock and Seymour [1] proved that for any connected graph G, allowing recontamination cannot reduce the search number. Thus, we only need to consider monotonic search strategies. Megiddo et al. [9] showed that determining the search number of a graph G is NP-hard. They also gave a linear 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. [10] proposed a linear time algorithm to compute the optimal search strategy of trees.

Search numbers are closely related to several other important graph parameters, such as vertex separation and pathwidth. 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 $xy \in E(G)$, $L(x) \leq i$ and $L(y) > i\}$. The *vertex separation of G with respect to L*, denoted by $\mathrm{vs}_L(G)$, is defined as $\mathrm{vs}_L(G) = \max\{|V_L(i)| : 1 \leq i \leq |V(G)|\}$. 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 $\mathrm{vs}_L(G) = \mathrm{vs}(G)$. Kinnersley [6] showed that $\mathrm{vs}(G)$ equals the pathwidth of G. Ellis et al. [2] proved that $\mathrm{vs}(G) \leq s(G) \leq \mathrm{vs}(G)+2$ for any connected undirected graph G. They gave a transformation called 2-expansion from G to G' such that $\mathrm{vs}(G') = s(G)$. They also described an algorithm for trees to compute the vertex separation in linear time. Based on this algorithm, Ellis and Markov [3] gave an $O(n \log n)$ algorithm for computing the vertex separation and the optimal layout of a unicyclic graph.

The rest of this paper is organized as follows. In Section 2, we give definitions and notation. In Section 3, we present a linear time algorithm for computing the search number and the optimal search strategy for a tree by applying the labeling method. In Section 4, we improve Ellis and Markov's algorithm from $O(n \log n)$ to $O(n)$ for computing the vertex separation and the optimal layout of a unicyclic graph. In Section 5, we show how to compute the search number of a k-ary cycle-disjoint graph. In Section 6, we investigate approximation algorithms for computing the search number of a cycle-disjoint graph.

2 Preliminaries

All graphs in this paper are finite without loops and multiple edges. A *rooted tree* is a tree with one vertex designated as the root of the tree. We use $T[r]$ to denote a rooted tree T with root r. For any two vertices v_1 and v_2 in $T[r]$, if there is a path from r to v_2 that contains v_1, then we say v_2 is a *descendant* of v_1; specifically, if v_2 is adjacent to v_1, we say v_2 is a *child* of v_1. Each vertex of $T[r]$ except r is a descendant of r. For any edge with end vertices u and v, if v is the child of u, then we orient this edge with the direction from u to v. This edge is denoted by (u, v). After this orientation, we obtain a directed rooted tree $T[r]$ such that the in-degree of r is 0 and the in-degree of any other vertex is 1. For any vertex v of $T[r]$, the subtree induced by v and all its descendant vertices is called the *vertex-branch* at v, denoted by $T[v]$. $T[r]$ can be considered as a vertex-branch at r. For any directed edge (u, v), the graph $T[v] + (u, v)$ is called the *edge-branch* of (u, v), denoted by $T[uv]$. $T[uv]$ is also called an *edge-branch at u*.

A vertex-branch $T[x]$ is said to be *k-critical* if $s(T[x]) = k$ and there are exactly two edge-disjoint edge-branches in $T[x]$ such that they share a common vertex and each has search number k. This common vertex is called a *k-critical vertex*. An edge-branch $T[xy]$ is k-critical if $s(T[xy]) = k$ and $T[xy]$ contains a

k-critical vertex-branch. We use k^+ to denote a *critical element*, where k is a positive integer. The value of k^+, denoted as $|k^+|$, is equal to k.

Let $T[r]$ be a rooted tree and v be a vertex in $T[r]$. If $s(T[v]) = s_1$ and $T[v]$ is s_1-critical, let v_1 be the s_1-critical vertex in $T[v]$ and let $T[v, v_1]$ denote the subtree obtained by deleting all edges and vertices (except v_1) of $T[v_1]$ from $T[v]$. If $s(T[v, v_1]) = s_2$ and $T[v, v_1]$ is s_2-critical, let v_2 be the s_2-critical vertex in $T[v, v_1]$ and let $T[v, v_1, v_2]$ denote the subtree obtained by deleting all edges and vertices (except v_2) of $T[v_2]$ from $T[v, v_1]$. Repeat this process until we first encounter a subtree $T[v, v_1, \ldots, v_k]$ that is a single vertex v or whose search number is equal to s_{k+1} and which is not s_{k+1}-critical. If $T[v, v_1, \ldots, v_k]$ is a single vertex, then the label of v, denoted by $L(v)$, is defined by $(s_1^+, s_2^+, \ldots, s_k^+)$; otherwise, the label $L(v)$ is defined by $(s_1^+, s_2^+, \ldots, s_k^+, s_{k+1})$. Specifically, if $s(T[v]) = s_1 > 0$ and $T[v]$ is not s_1-critical, then the label $L(v)$ is defined by (s_1). Let (u, v) be an edge in $T[r]$. If $s(T[uv]) = s_1$ and $T[uv]$ is s_1-critical, let v_1 be the s_1-critical vertex in $T[uv]$ and let $T[uv, v_1]$ denote the subtree obtained by deleting all edges and vertices (except v_1) of $T[v_1]$ from $T[uv]$. If $s(T[uv, v_1]) = s_2$ and $T[uv, v_1]$ is s_2-critical, let v_2 be the s_2-critical vertex in $T[uv, v_1]$ and let $T[uv, v_1, v_2]$ denote the subtree obtained by deleting all edges and vertices (except v_2) of $T[v_2]$ from $T[uv, v_1]$. Repeat this process until we first encounter a subtree $T[uv, v_1, \ldots, v_k]$ whose search number is equal to s_{k+1} and which is not s_{k+1}-critical. The label of uv, denoted by $L(uv)$, is defined by $(s_1^+, s_2^+, \ldots, s_k^+, s_{k+1})$. Specifically, if $s(T[uv]) = s_1 > 0$ and $T[uv]$ is not s_1-critical, then the label of uv is defined by (s_1). Both vertex labels and edge labels have the following property.

Lemma 1. *For a labeled tree, each vertex label or edge label consists of a sequence of strictly decreasing elements such that each element except the last one must be a critical element.*

3 Labeling Method for Trees

From [2], we know that the search number of a tree can be found in linear time by computing the vertex separation of the 2-expansion of the tree. From [9], we know that the search number of a tree can also be found in linear time by using a hub-avenue method. However, in order to apply the labeling method proposed in [2] to compute search numbers of other special graphs (refer to the full version [11] of this paper), we modify this method so that it can compute the search number of a tree directly.

For a tree T, if we know the search number of all edge-branches at a vertex in T, then $s(T)$ can be computed from combining these branches' information.

Algorithm. SEARCHNUMBER($T[r]$)
Input: A rooted tree $T[r]$.
Output: $s(T[r])$.

1. Assign label (0) to each leaf (except r if r is also a leaf) and assign label (1) to each pendant edge in $T[r]$.

```
2. if there is an unlabeled vertex $v$ whose all out-going edges have been labeled,
   then compute the label $L(v)$;
        if $v$ has an in-coming edge $(u, v)$, then
             computer the label $L(uv)$;
        else stop and output the value of the first element in the label $L(v)$.
   repeat Step 2.
```

Because a rooted tree $T[r]$ has a unique root r, every vertex except r has a unique in-coming edge and r has no in-coming edge. Thus, if v has no in-coming edge at line 3 of Step 2, then v must be the root r and the value of the first element in its label is equal to the search number of $T[r]$. We can prove the following results for rooted trees.

Lemma 2. *For a rooted tree $T[r]$ and a vertex v in $T[r]$, let $v_1, v_2, \ldots, v_k$ be all the children of v. Let $a = \max\{s(T[vv_i]) \mid 1 \leq i \leq k\}$ and b be the number of edge-branches with search number a.*
(i) *If $b \geq 3$, then $s(T[v]) = a + 1$.*
(ii) *If $b = 2$ and no edge-branch at v is a-critical, then $s(T[v]) = a$ and $T[v]$ is a-critical.*
(iii) *If $b = 2$ and at least one branch at v is a-critical, then $s(T[v]) = a + 1$.*
(iv) *If $b = 1$ and no branch at v is a-critical, then $s(T[v]) = a$.*
(v) *If $b = 1$ and $T[vv_j]$ is a-critical, let u be the a-critical vertex in $T[vv_j]$, and let $T[v, u]$ be the subtree formed by deleting all edges of $T[u]$ from $T[v]$. If $s(T[v, u]) = a$, then $s(T[v]) = a + 1$; and if $s(T[v, u]) < a$, then $s(T[v]) = a$ and $T[v]$ is a-critical.*

In Step 2 of the algorithm SearchNumber($T[r]$), let $v_1, v_2, \ldots, v_k$ be all the children of v. Each label $L(vv_i)$ contains the structure information of the edge-branch $T[vv_i]$. For example, if $L(vv_i) = (s_1^+, s_2^+, \ldots, s_m^+, s_{m+1})$, it means $T[vv_i]$ has a s_1-critical vertex u_1, $T[vv_i, u_1]$ has a s_2-critical vertex u_2, $\ldots$, $T[vv_i, u_1, \ldots, u_{m-1}]$ has a s_m-critical vertex u_m, and $T[vv_i, u_1, \ldots, u_m]$ has search number s_{m+1} and it is not s_{m+1}-critical. From Lemma 2, we can compute $L(v)$ that contains the structure information of the vertex-branch $T[v]$ by using the structure information of all edge-branches at v. Since the label of an edge (x, y) contains the information of the edge-branch $T[y] + (x, y)$, we can compute $L[xy]$ from $L[y]$. By using appropriate data structures for storing labels, each loop in Step 2 can be performed in $O(s(T[vv']) + k)$ time, where $T[vv']$ is the edge-branch that has the second largest search number among all edge-branches at v and k is the number of children of v. By using a recursion to implement Step 2 of SearchNumber($T[r]$), we can prove the following result.

Theorem 1. *If n is the number of vertices in a tree T, then the running time of computing $s(T)$ is $O(n)$.*

After we find the search number, we can use the information obtained in Algorithm Search-Number($T[r]$) to compute an optimal monotonic search strategy in linear time.

4 Unicyclic Graphs

Ellis and Markov [3] proposed an $O(n \log n)$ algorithm to compute the vertex separation of a unicyclic graph. 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 [3]. Their algorithm consists of three functions: main, vs_uni and vs_reduced_uni (see Fig. 28, 29 and 30 in [3] for their descriptions).

Let U be a unicyclic graph and e be a cycle edge of U. In function main, it 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 invokes 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 computes the vertex separation of $T_1[a]$ and $T_1[b]$ and this takes $O(n)$ time. Thus, the running time of vs_reduced_uni is also $O(n \log n)$.

We will modify all three functions. The main improvements of our algorithm are to preprocess the input of both vs_uni and vs_reduced_uni so that we can achieve $O(n)$ running time. The following is our improved algorithm, which computes the vertex separation and the corresponding layout for 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($\alpha$, 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 $\text{vs}(T')$;
         if $\text{vs}(T') = k$, 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 $\text{vs}(T_1[a])$, $\text{vs}(T_1[b])$, $\text{vs}(T_2[c])$ and $\text{vs}(T_2[d])$;
         /* Assume that $\text{vs}(T_1) \geq \text{vs}(T_2)$. */
         /* 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 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 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 3. *Let U be a unicyclic graph, e be a cycle edge and r be a cycle vertex in U. Let $T[r]$ denote the 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.*

The correctness of the modified algorithm follows from the analysis in Sections 4 and 5 in [3]. We now compare the two algorithms. In our main_modified function, if the condition of the *while-loop* is satisfied, then by Lemma 3, 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 [3]. 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 [3] 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 3, 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 $O(n)+O(\log n)$. Hence, the total running time of the modified algorithm is $O(n)$. Therefore, we have the following theorem.

Theorem 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 [2], the search number of G is equal to the vertex separation of the 2-expansion of G. From Theorem 2, we have the following result.

Corollary 1. *For a unicyclic graph G, the search number and the optimal search strategy of G can be computed in linear time.*

5 *k*-Ary Cycle-Disjoint Graphs

A graph G is called a *cycle-disjoint graph (CDG)* if it is connected and no pair of cycles in G share a vertex. 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. In this section, we will show how to compute the search numbers of k-ary CDGs. Similar to [3], we have the following lemmas.

Lemma 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 $s(G) \geq k+1$.*

Lemma 5. *For a connected graph G, let $C = v_1v_2 \dots v_mv_1$ be a cycle in G such that each v_i $(1 \leq i \leq m)$ connects to a connected subgraph X_i by a bridge. If $s(X_i) \leq k, 1 \leq i \leq m$, then $s(G) \leq k+2$.*

Lemma 6. *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 $(1 \leq i \leq 5)$ connects to a connected subgraph X_i by a bridge. If $s(X_i) \geq k, 1 \leq i \leq 5$, then $s(G) \geq k+2$.*

Lemma 7. *For a connected graph G, let $C = v_1v_2v_3v_4v_1$ be a 4-cycle in G such that each v_i $(1 \leq i \leq 4)$ connects to a connected subgraph X_i by a separation edge. If $s(G) = k+1$ and $s(X_i) = k, 1 \leq i \leq 4$, then for any optimal monotonic search strategy of G, the first cleared vertex and the last cleared vertex must be in two distinct graphs $X_i + v_i$, $1 \leq i \leq 4$.*

Lemma 8. *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'.*

For a vertex v in G, if $s(G) = k$ and there is no monotonic search strategy to clear G starting from or ending at v using k searchers, then we say that v is a *bad vertex* of G.

Lemma 9. *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 on C such that each v_i $(1 \leq i \leq 2)$ connects to a connected subgraph X_i by a bridge $v_i v_i'$. 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 10. *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 $(1 \leq i \leq 4)$ connects to a connected subgraph X_i by a bridge $v_i v_i'$. If $s(X_i) = k$, and v_i' is a bad vertex of X_i, $1 \leq i \leq 4$, then $s(G) \geq k + 2$.*

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

Theorem 3. *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 $h \geq 3$, 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 $h \geq 2$, then $s(T_3^h) = h$ and $s(G_3^h) = h + 1$.*
(iii) *If $k = 4$ and $h \geq 2$, then $s(T_4^h) = h$ and $s(G_4^h) = h + \lceil \frac{h}{2} \rceil$.*
(iv) *If $k \geq 5$ and $h \geq 2$, 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 the algorithm SearchNumber($T[r]$). Thus, we will only consider the search numbers of k-ary CDGs.

(i) The search number of G_2^h can be verified by a search strategy based on SearchStrategy($T[r]$).

(ii) We now prove $s(G_3^h) = h + 1$ by induction on h. Let $R = r_0 r_1 r_2 r_3 r_0$ 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 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 $r_1 r_2$; 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 $r_0 r_3$; 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 10, $s(G_4^3) \geq 5$ 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 10, 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 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 7, 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 5 and 6.

6 Approximation Algorithms

Megiddo et al. [9] 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 4. *Given a CDG G, if T is a tree obtained by contracting each cycle of G into a vertex, then $s(T) \leq s(G) \leq 2s(T)$.*

Corollary 2. *For any CDG, there is a linear time approximation algorithm with approximation ratio 2.*

Lemma 11. *Let G be a CDG in which every cycle has at most three vertices with degree more than two. Let T be the tree obtained from G by contracting every cycle of G into a vertex. If the degree of each cycle vertex in G is at most three, then $s(G) \le s(T) + 1$.*

Let $S = (a_1, \ldots, a_k)$ be an optimal monotonic search strategy for a graph. The *reversal* of S, denoted as S^R, is defined by $S^R = (\overline{a}_k, \overline{a}_{k-1}, \ldots, \overline{a}_1)$, 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 "move the searcher from v to u along the edge vu" and the action "move the searcher from u to v along the edge uv" are converse with each other.

Lemma 12. *If S is an optimal monotonic search strategy of a graph G, then S^R is also an optimal monotonic search strategy of G.*

Lemma 13. *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.*

Theorem 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. 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 $s(G^*) = 1$, then G^* is a single vertex or a path, and $\lceil \Delta(G^*)/2 \rceil = 1$. Suppose that G^* is the path $v_1 v_2 \ldots v_m$ 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 13, 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. Suppose that 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 $s(T_i) \leq n$. Thus, $s(X(T_i)) \leq \max_i s(X_i) + \lceil \Delta(T_i)/2 \rceil n \leq \max_i s(X_i) + \lceil \Delta(G^*)/2 \rceil n$. First, we place a searcher on each a_i, $1 \leq i \leq \lceil \deg(v)/2 \rceil$. 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, $1 \leq i \leq \lceil \deg(v)/2 \rceil$. 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 $\max_i s(X_i) + \lceil \Delta(G^*)/2 \rceil n + \lceil \deg(v)/2 \rceil \leq \max_i s(X_i) + \lceil \Delta(G^*)/2 \rceil (n+1)$ searchers.

Case 2. G^* has an avenue $v_1 v_2 \ldots v_t$, $t > 1$. 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 $X(v_i)$, $0 \leq i \leq t+1$, be the subgraph of G that corresponds to v_i. For $0 \leq i \leq t$, let b_i be the vertex shared by $X(v_i)$ and $X(v_{i+1})$. For $1 \leq i \leq t$, 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, 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.

References

1. Bienstock, D., Seymour, P.: Monotonicity in graph searching. Journal of Algorithms 12, 239–245 (1991)
2. Ellis, J., Sudborough, I., Turner, J.: The vertex separation and search number of a graph. Information and Computation 113, 50–79 (1994)
3. Ellis, J., Markov, M.: Computing the vertex separation of unicyclic graphs. Information and Computation 192, 123–161 (2004)
4. Fellows, M., Langston, M.: On search, decision and the efficiency of polynomial time algorithm. In: 21st ACM Symp. on Theory of Computing, pp. 501–512. ACM Press, New York (1989)
5. Frankling, M., Galil, Z., Yung, M.: Eavesdropping games: A graph-theoretic approach to privacy in distributed systems. Journal of ACM 47, 225–243 (2000)
6. Kinnersley, N.: The vertex separation number of a graph equals its path-width. Information Processing Letters 42, 345–350 (1992)
7. Kirousis, L.M., Papadimitriou, C.H.: Searching and pebbling. Theoretical Computer Science 47, 205–218 (1986)
8. LaPaugh, A.S.: Recontamination does not help to search a graph. Journal of ACM 40, 224–245 (1993)
9. Megiddo, N., Hakimi, S.L., Garey, M., Johnson, D., Papadimitriou, C.H.: The complexity of searching a graph. Journal of ACM 35, 18–44 (1988)
10. Peng, S., Ho, C., Hsu, T., Ko, M., Tang, C.: Edge and node searching problems on trees. Theoretical Computer Science 240, 429–446 (2000)
11. Yang, B., Zhang, R., Cao, Y.: Searching cycle-disjoint graphs. Technical report CS-2006-05, Department of Computer Science, University of Regina (2006)