Searching Trees with Sources and Targets*

Chris Worman and Boting Yang

Department of Computer Science, University of Regina
{worman2c,boting}@cs.uregina.ca

Abstract. We consider a new pursuit-evasion problem on trees where a subset of vertices, called *sources*, are initially occupied by searchers. We also consider the scenario where some of the searchers must end their search at certain vertices called *targets*. We incrementally consider such problems, first considering only sources, then only targets, and finally we consider the case where there are both sources and targets. For each case we provide a polynomial-time algorithm for computing the search number, i.e. the minimum number of searchers required to clear the tree, and an optimal search strategy. We also demonstrate that each search model is monotonic, i.e. for each case their exists an optimal search strategy such that the set of cleared edges grows monotonically as the search progresses.

1 Introduction

Imagine a scenario where a group of police officers is attempting to capture a fugitive who is hiding in a building. The police officers could enter the building and hope to capture the fugitive by exploring the building in an ad hoc manner. A better approach is to systematically explore the building to ensure the fugitive is captured. If we can guarantee that we can capture the fugitive by carefully choosing our search strategy, the following question becomes relevant: "What is the minimum number of police officers required to capture the fugitive?" This question motivates so-called *graph searching* problems, which are pursuit-evasion problems that take place on graphs.

In a typical graph searching problem introduced by Megiddo et al. [4], a group of searchers must capture an exceedingly fast and clever fugitive that is hiding on a graph. The graph is meant to represent some real-world domain, such as the hallways and rooms in a building. The searchers proceed by clearing edges, i.e. visiting edges to guarantee that the fugitive is not on the edge; initially all edges are *dirty*. Three searcher actions are allowed: (1) place a searcher on a vertex in the graph, (2) remove a searcher from a vertex, and (3) slide a searcher along an edge from one end vertex to the other. An edge uv is *cleared* when either (1) at least two searchers are located on a vertex u, and one of them slides along uv, or (2) a searcher is located on u, and all edges incident on u, except uv, are clear and the searcher slides along uv. A *search strategy* is a sequence of searcher actions that

* Research was supported in part by NSERC.

F.P. Preparata, X. Wu, and J. Yin (Eds.): FAW 2008, LNCS 5059, pp. 174–185, 2008.
© Springer-Verlag Berlin Heidelberg 2008

result in every edge in the graph being cleared. Researchers are often interested in computing the *search number* of a graph G, denoted $s(G)$, which is the minimum number of searchers required to clear G. A search strategy is called optimal if at each step it uses at most $s(G)$ searchers. Another important consideration for searching problems is that of monotonicity. Define A_i to be the set of edges that are clear after the ith action of some searching strategy. We define $A_0 = \emptyset$. A search strategy $S = \{m_1, m_2, \ldots, m_r\}$ is called *monotonic* if $A_{i-1} \subseteq A_i$, for all $1 \leq i \leq r$. A searching problem is in turn called *monotonic* if there exists an optimal monotonic search strategy for each instance of the searching problem. For non-monotonic search strategies, if a searcher is removed or slides from a vertex such that the resulting graph contains a path, which is not occupied by any searchers, connecting a dirty edge to a clear edge then the clear edge becomes *recontaminated* instantaneously because the fugitive moves exceedingly fast.

Graph searching was first studied by Parsons [5], who studied a continuous version of the graph searching problem. This model was subsequently discretized by Megiddo et al. [4], who showed that deciding the search number of a graph is NP-hard [4]. The monotonicity result of LaPaugh [3] demonstrates that deciding the search number is in NP, and hence deciding the search number of a graph is NP-complete. Megiddo et al. [4] also provided an $O(n)$ time algorithm for computing the search number of a tree that can be extended to compute an optimal search strategy in $O(n \log n)$ time. Subsequently, Peng et al. [6] improved this result by giving an $O(n)$ time algorithm for computing an optimal search strategy of a tree. The search number of a graph has been shown to be related to some important graph parameters, such as vertex separation number and pathwidth. Other graph searching models have been studied; see [1,2] for surveys.

All the searching problems discussed in this paper take place on trees. In the problems that we study, two disjoint subsets of vertices of a tree T have been identified: the *sources*, denoted V_s, and the *targets*, denoted V_t. We use V_s and V_t to define a constrained graph searching problem as follows. At the beginning of the search, each source vertex is occupied by exactly one searcher. Furthermore, for the entire duration of the search strategy, each source vertex remains clear, i.e. is occupied by a searcher or all incident edges are clear. Also, once a searcher occupies a vertex $v \in V_t$, v must be occupied by at least one searcher for the remainder of the search strategy. We refer to such searching problems as *Source Target Searching (STS)* problems. In an STS problem, we call the searchers that were initially on sources *starting searchers*. All other searchers are called *additional searchers*. During the progression of a search strategy, a starting searcher may be removed. A starting searcher is called *free* when it is not currently occupying a vertex on T.

One of our primary motivations for studying STS problems comes from an algorithm development point of view: our algorithms for STS problems can be used as subroutines for searching algorithms for other classes of graphs, such as cycle-disjoint graphs [7], in which lots of induced subgraphs are trees. In this application, the starting searchers are placed as a result of some other algorithm. The algorithms presented herein can then be used to complete the search while

still ensuring that the already cleared portions of the graph remain clear. Target vertices represent vertices where certain searchers must terminate. In this application, the target vertices represent portals to other parts of the graph that are contaminated, but that we are not willing to visit at this point in the search. Thus a target searching algorithm can be used as a subroutine for searching algorithms for other classes of graphs that first clear an induced subgraph that is a tree, and then clear the remaining parts of the graph.

Another motivation for studying STS problems comes from scenarios where the graph contains vertices that are portals to uncontaminated or already cleared areas of the graph. For example, consider the scenario where the graph models a network of city streets where a fugitive is hiding. In this case, we may be able to restrict our search to a particular neighborhood. We can place starting searchers at intersections connecting the neighborhood with the rest of the city. These intersections are modeled by the source vertices in our searching model. The source vertices are initially protected by the starting searchers, and remain protected throughout the search, ensuring the fugitive remains in the neighborhood that we are searching. Extending this example to target vertices, we may wish to search for the fugitive "neighborhood by neighborhood." In this scenario, we first search a particular neighborhood, ensuring that when a searcher encounters an intersection leading another neighborhood, this searcher stays at the intersection until the original neighborhood is cleared. These intersections are modeled by the target vertices in an instance of the STS problem. Thus STS problems can be seen as a particular type of subgraph searching.

In Section 2 we describe an $O(|V_s|n)$ time algorithm for computing the search number of a tree when the tree contains sources, but does not contain any targets. In Section 3 we demonstrate an inverse relationship between searching with sources and searching with targets. We exploit this relationship in order to use the algorithm from Section 2 to compute the search number of a tree that contains only targets in $O(|V_t|n)$ time. In Section 4 we combine these results and extend the algorithm from Section 2 so that we can compute the search number of a tree with both sources and targets in $O(|V_s||V_t|n)$ time. In all cases, we show that the algorithms only output monotonic search strategies, thus establishing the monotonicity of each search model. Due to space constraints, some details have been omitted.

2 Searching Trees with Sources

In this section we consider the STS problem on trees that contain only sources. We call this type of searching *Source Searching*, or simply *SS*. Let T be a tree and $V_s \subseteq V(T)$ be a set of source vertices such that each vertex in V_s is initially occupied by a starting searcher. We define the *source search number* with respect to T an V_s, denoted $ss(T, V_s)$, as the minimum number of additional searchers required to clear T. We deem a vertex v to be *clear* if all edges incident with v are clear. A searcher occupying a vertex v in V_s is *moveable* if all but one edges incident with v are clear. We can formally state our searching problem as follows:

Source Searching (SS)

Instance: A tree T and a set of source vertices $V_s \subseteq V(T)$, such that each $v \in V_s$ is occupied by exactly one searcher.

Question: Is there a search strategy of T using k additional searchers such that each vertex from V_s remains clear during the entire search strategy?

Recall that in the SS problem, all edges are initially dirty. In order to make our analysis more simple, we actually describe an algorithm for a more general searching problem, which we call *Partial-Clear Source Searching (PSS)*, which allows certain subtrees to be clear at the beginning of the problem. In order to define this new searching problem, we require some notation. Given a tree T and a subset of vertices $V_s \subseteq V(T)$, define $T \ominus V_s$ to be the set of maximal induced subtrees $\{T_1, T_2, ..., T_k\}$ of T such that if $v \in V_s$ and $v \in T_i$, then v is a leaf in T_i, and $\bigcup\{T_i\} = T$, $1 \leq i \leq k$.

Partial-Clear Source Searching (PSS)

Instance: A tree T with a set of cleared edges $E_c \subseteq E(T)$, and a set of source vertices $V_s \subseteq V(T)$, such that each $v \in V_s$ is occupied by exactly one searcher and each subtree in $T \ominus V_s$ is either completely clear or completely dirty.

Question: Is there a search strategy of T using k additional searchers such that each vertex from V_s remains clear during the entire search strategy?

The associated search number for this searching problem is denoted $pss(T, V_s, E_c)$. One can see that an instance of the SS problem is a valid instance of the PSS problem with $E_c = \emptyset$. Moreover, since the two search problems have the same objective, any algorithm for the PSS problem can be used to solve the SS problem. Thus, for the remainder of this section we focus on the PSS problem, keeping in mind that the results also hold for the SS problem.

Our algorithm for computing $pss(T, V_s, E_c)$ recursively performs two major steps in order to clear T: firstly, the algorithm calls a subroutine called `Reposition`, which moves some of the searchers occupying sources, and secondly the algorithm clears a specially chosen subtree of T. Specifically, the algorithm will clear the dirty subtree from $T \ominus V_s$ with the smallest search number. Then a recursive call is made in order to search the remaining tree. The algorithm is formally stated in Figures 1 and 2. The algorithm makes use of a *global* variable f that records how many of the starting searchers have been removed thus far, i.e. f is equal to the number of free starting searchers. Thus, $f = 0$ initially. The algorithm computes a value s_{max}, which we claim is equal to $pss(T, V_s, E_c)$.

Now we turn our attention to proving the correctness of our algorithm. Intuitively, our proof approach is the following. Firstly, Lemma 1 essentially proves that the actions performed by the `Reposition` algorithm are part of some optimal strategy. Secondly, Lemma 2 essentially shows that $pss(T, V_s, E_c)$ is at least as large as the search number of the first subtree that the algorithm clears. Since our algorithm is recursive, once these two Lemmas are shown, the correctness of our algorithm follows. We begin with the following observation:

```
Algorithm Reposition(T, V_s, E_c)

1. While there exists a moveable searcher λ on vertex v ∈ V_s:
   (a) Let vu be the only dirty edge incident with v.
   (b) Slide λ from v to u, clearing the edge vu, and V_s ← (V_s − {v}) ∪ {u},
       E_c ← E_c ∪ {vu}.
   (c) If u contains two searchers then remove λ and f ← f + 1.
2. While there exists a searcher λ on a clear vertex v ∈ V_s:
   (a) Remove λ and V_s ← V_s − {v}.
   (b) f ← f + 1.
```

Fig. 1. The algorithm for repositioning the starting searchers

```
Algorithm ST-S(T, V_s, E_c) (Search Tree with Sources)

1. If T is clear then return 0.
2. Call Reposition(T, V_s, E_c).
3. Compute T ⊖ V_s, discarding all clear subtrees, to obtain {T_1, T_2, ..., T_k}.
   Without loss of generality, suppose that s(T_1) ≤ s(T_i), for 2 ≤ i ≤ k.
4. Clear all edges of T_1 using s(T_1) searchers and E_c ← E_c ∪ E(T_1).
5. s_max ← max{s(T_1) − f, ST-S(T, V_s, E_c)}.
6. Return s_max.
```

Fig. 2. The algorithm for computing $pss(T, V_s, E_c)$

Lemma 1. *Let (T, V_s, E_c) be an instance of the PSS problem. If V_s' and E_c' are the results of running the* `Reposition`(T, V_s, E_c) *algorithm, then* $pss(T, V_s', E_c') \leq pss(T, V_s, E_c) + f$*, where f is the number of starting searchers removed by the* `Reposition`(T, V_s, E_c) *algorithm.*

Proof. Let S be an optimal PSS strategy for (T, V_s, E_c). We construct a PSS strategy S' for (T, V_s', E_c') that does exactly what S does, except in a few cases. Before describing these cases, we introduce some notation and make some observations. Let λ_v be the starting searcher located on vertex v. Define $(v, v^1, v^2, ..., v^l)$ to be the path that λ_v slides along during the execution of the `Reposition`(T, V_s, E_c) algorithm. From the condition of the while loop in step 1 of the `Reposition` algorithm, we know that every v^i, $1 \leq i \leq l-1$, has degree two. Notice that the path $(v, v^1, ..., v^l)$ is dirty in (T, V_s, E_c) and has been cleared in (T, V_s', E_c').

Now we describe the changes made to S in order to obtain S'. The following change ensures that λ_v is located on T so that it can mimic the movements of λ_v in S.

1. For each λ_v that was removed by the `Reposition` algorithm, S' begins by placing λ_v on v^l.

The next change ensures that $v^l \in V_s'$ does not become incident with a dirty edge while it is unoccupied in S:

2. Whenever a searcher λ_* is placed on or slides to a vertex $v^i \in \{v, v^1, v^2, ..., v^{l-1}\}$, in S' we place λ_* on v^l, unless in S' λ_* is already occupying v^l, in which case in S' we do nothing. If in S λ_* is subsequently removed from v^i or slides to a vertex not in $\{v, v^1, v^2, ..., v^{l-1}, v^l\}$, then λ_* is removed from v^l, placed on v^i, and then λ_* performs this action.

Let $a_i \in S$ be the first action that either removes or slides λ_v. In S', prior to performing a_i, we do the following:

3. Remove λ_v from v^l, place λ_v on v, and then perform a_i.

Notice that since S clears T, then S' clears T. In order to complete the proof we must show that S' uses at most $pss(T, V_s, E_c) + f$ additional searchers and that all members of V_s' remain clear during the progression of S'. Notice that in S', the only new searchers that are added are those that were removed by the `Reposition` algorithm (see change (1) above), and there are exactly f searchers removed by the `Reposition` algorithm. Hence S' uses $pss(T, V_s, E_c) + f$ searchers.

All that remains is to demonstrate that all members of V_s' remain clear during the progression of S'. We can focus on the actions introduced in the changes given above since in all other cases, S' performs the same actions as in S. Specifically, we must consider the case where λ_v is removed from v^l in change (3), and when λ_* is removed from v^l (and then placed on v^i) in change (2). In either case, we have that in S there is no searcher occupying a vertex from $\{v, v^1, v^2, ..., v^{l-1}\}$ when λ_v performs a_i, otherwise this searcher would have been placed on v^l in change (2). Thus in S the dirty edge e is connected to $v \in V_s$ via a path containing no searchers, and v is unoccupied, which is a contradiction. □

In the following lemma, T_1 is defined as in step 3 of the `ST-S` algorithm; that is, after the call to the `Reposition` algorithm.

Lemma 2. *For any instance (T, V_s, E_c) of the PSS problem, $pss(T, V_s) \geq s(T_1) - f$, where T_1 is the subtree computed after steps 1-3 of the* `ST-S(`T, V_s, E_c`)` *algorithm, and f is the number of free starting searchers removed during the call to* `Reposition(`T, V_s, E_c`)` *in step 2 of the* `ST-S(`T, V_s, E_c`)` *algorithm.*

Proof. Let V_s' and E_c' be the results of running the `Reposition(`T, V_s, E_c`)` algorithm. By Lemma 1, we have that $pss(T, V_s', E_c') - f \leq pss(T, V_s, E_c)$, and hence it suffices to show that $pss(T, V_s', E_c') \geq s(T_1)$. Let S^* be any optimal PSS strategy for the instance (T, V_s', E_c'). Define T^* to be the first subtree from $\{T_1, T_2, ..., T_k\}$ (as defined in the algorithm) that S^* completely clears[1]. Recall that $s(T_1) \leq s(T_i)$, for $2 \leq i \leq k$. If S^* only uses additional searchers to clear T^* then $pss(T, V_s', E_c') \geq s(T*) \geq s(T_1)$ and the Lemma holds. Thus we only need to consider the case where S^* uses at least one starting searcher to aid in clearing T^*. In what follows, we show that this implies a contradiction by demonstrating that for each starting searcher used to clear T^*, there exists a corresponding additional searcher on the tree T.

[1] S^* may have cleared other edges of T prior to completely clearing T^*.

We begin by demonstrating a property of S^* that we will use in the rest of the proof. Suppose that at some point during S^*, a source vertex $v \in V_s$ is occupied, but is not occupied by the original starting searcher λ_v that occupied v in the instance (T, V_s', E_c'). We can adjust S^* as follows: if λ_v is removed from v, then either all edges incident with v are clear, in which case no searcher needs to occupy v for the remainder of S^*, or v is occupied by another searcher, and we can remove this searcher instead of removing λ_v. If λ_v slides off v at some point, then after it slides off either all incident edges are clear, in which case no searcher needs to occupy v for the remainder of S^*, or there is another searcher occupying v, and we can slide this searcher instead of λ_v. So without loss of generality, during the progression of S^*, if a source vertex $v \in V_s$ is occupied, then it is occupied by λ_v, where λ_v is the starting searcher that occupies v in the instance (T, V_s', E_c'). We use this fact throughout the remainder of the proof.

Define $\{\lambda_1, \lambda_2, ..., \lambda_b\}$ to be the set of starting searchers occupying T^* at some moment during S^*. We define a procedure for associating a unique additional searcher with each starting searcher in $\{\lambda_1, \lambda_2, ..., \lambda_b\}$, but first we require some definitions. We begin by defining two sets of source vertices V^* and U^*, which form a partition of the currently unoccupied source vertices. Define $V^* = \{v_1, v_2, ..., v_b\}$ to be the set of source vertices that $\{\lambda_1, \lambda_2, ..., \lambda_b\}$ initially occupied. Define $U^* = \{u_1, u_2, ..., u_k\}$ to be the set of unoccupied source vertices whose original starting searchers are not currently occupying T^*. For a subset of source vertices $V_s' \subseteq V_s$, define $P(V_s') = \{T_{i_1}, T_{i_2}, ..., T_{i_k}\}$ to be the subset of trees from $\{T_1, T_2, ..., T_k\}$ such that for each $T_i \in P(V_s')$ there exists a $v \in V_s'$ such that v is a leaf in T_i.

We define two graphs G and H as folllows. Define $G = (V(G), E(G))$ as $V(G) = V^* \cup P(V^*)$ and $\{v, T_i\} \in E(G)$ if v is a leaf in T_i. Define $H = (V(H), E(H))$ as $V(H) = U^* \cup P(U^*)$, and $\{u, T_i\} \in E(G)$ if u is a leaf in T_i. One can easily show that both G and H are forests since T is a tree.

Now consider the following procedure for assigned a unique additional searcher, which is currently occupying T, to each of $\{\lambda_1, \lambda_2, ..., \lambda_b\}$. Let T_i be a leaf of G (notice that leaves in G cannot be source vertices since (T, V_s', E_c') is the result of running the `Reposition` algorithm). Let $v_j \in V^*$ be the parent of T_i in G. Since initially each source vertex is adjacent to two or more dirty edges, and since T^* is the first tree from $\{T_1, T_2, ..., T_k\}$ to be cleared, there must be a searcher, say λ_*, in T_i which is protecting v_i from becoming adjacent with a dirty edge.

If λ_* is an additional searcher, then we associate λ_* with λ_j, and remove v_j and T_i from G. Otherwise, the searcher located in T_i is a starting searcher. By our assumption about S^* (i.e. during the progression of S^*, if a source vertex $v \in V_s$ is occupied, then it is occupied by λ_v, where λ_v is the starting searcher that occupies v in the instance (T, V_s', E_c')), this means that λ_*'s original source vertex v_* is currently unocuppied, and hence $v_* \in V(H)$. Since (T, V_s', E_c') is result of running the `Reposition` algorithm, the vertex v_* is adjacent to at least two trees T_k and T_l in H. Since T^* is the first tree from $\{T_1, T_2, ..., T_k\}$ to be cleared, there must be a searcher, say λ_k, in T_k. If λ_k is an additional searcher, then associate λ_k with λ_j, and we remove T_k and λ_* from H, and remove v_j and

T_i from G. If λ_k is not an additional searcher, then we can recursively consider the trees adjacent to v_k in H, and then consider the searcher that must be located in one of the subtrees incident to v_k. Since H is a finite tree, this process must eventually stop when an additional searcher is found to be located in a subtree incident with some source vertex in H. When this additional searcher is found, we remove the appropriate source vertices and subtrees from H, remove v_j and T_i from G, and associate this additional searcher with λ_j.

We repeat the procedure given above for each leaf in G until G is empty. This procedure associates a unique additional searcher with each starting searcher currently occupying a vertex in T^*, as required. □

It is clear that the search strategy output by the `ST-S` algorithm actually clears T. Thus we can use induction and Lemma 2 to obtain one of our main results:

Theorem 1. *The value of s_{max} at the termination of the* `ST-S` *algorithm is equal to $pss(T, V_s, E_c)$.*

Theorem 2. *Let (T, V_s, E_c) be an instance of the PSS problem with $|V(T)| = n$. The source search number and the optimal search strategy of (T, V_s, E_c) can be computed in $O(|V_s|n)$ time.*

Theorem 3. *The PSS problem is monotonic.*

3 Searching Trees with Targets

In this section we study the problem where instead of having only *sources,* we have only *targets.*

Target Searching (TS)
Instance: A tree T with a set of target vertices $V_t \subseteq V(T)$.
Question: Is there a search strategy of T using k additional searchers such that once a searcher occupies a vertex $v \in V_t$, v remains occupied by at least one searcher for the remainder of the search?

Associated with this problem, we define the target search number, denoted $ts(T, V_t)$, to be the minimum number of searchers required to clear the tree T under the TS model. The TS problem can be seen as a kind of inversion of the SS problem. Given a search strategy S, define the *reverse* of a action $a \in S$, denote a^{-1}, as follows:

- If a is "slide λ from v to u", then a^{-1} is "slide λ from u to v".
- If a is "remove λ from v", then a^{-1} is "place λ on v".
- If a is "place λ on v", then a^{-1} is "remove λ from v".

Given a strategy $S = (a_1, a_2, ..., a_k)$, define the *inverse* of S, denoted S^{-1}, to be $S^{-1} = (a_k^{-1}, a_{k-1}^{-1}, \ldots, a_1^{-1})$.

Lemma 3. *Let (T, V_s) be an instance of the SS problem and (T, V_t) be an instance of the TS problem such that $V_s = V_t$. Then S is a monotonic SS strategy for (T, V_s) if and only if S^{-1} is a monotonic TS strategy for (T, V_t).*

Proof. (sketch) Let $V' = V_s (= V_t)$. If S is a TS strategy for (T, V'), then it is clear that S^{-1} ends with V' occupied by searchers. Conversely, if S^{-1} is a TS strategy for (T, V'), then $(S^{-1})^{-1} = S$ begins with V' occupied by searchers. To complete the proof, we must show that S monotonically clears T if and only if S^{-1} monotonically clears T. Since S and S^{-1} are inverses, it suffices to demonstrate the following property: if S clears T monotonically then S^{-1} clears T monotonically. For any sequence of actions $(b_1, b_2, ..., b_l)$, define $A(b_1, b_2, \ldots, b_l)$ to be the set of edges cleared by a sequence of actions $(b_1, b_2, \ldots, b_l)$. We want to show that $A(a_k^{-1}, a_{k-1}^{-1}, \ldots, a_1^{-1}) = A(a_1, a_2, ..., a_k)$.

We proceed by induction on the number of actions completed by S^{-1}. Initially no edges are cleared in the TS model, and the last action of a TS model is a remove operation, and hence no edge is cleared by the last action in S. This provides the base case for induction.

Assume $A(a_k^{-1}, a_{k-1}^{-1}, \ldots, a_p^{-1}) = A(a_p, a_{p+1}, ..., a_k)$, for $p \leq k$. By carefully considering a_{p-1}^{-1} in S^{-1}, we can show that no recontamination occurs in S^{-1} unless it occurs in S, and the Lemma follows. □

In light of this observation, and Theorem 1, Theorem 3, and Theorem 2 from the previous section, we have the following two results:

Theorem 4. *Let (T, V_t) be an instance of the TS problem with $|V(T)| = n$. The target search number and the optimal search strategy of (T, V_t) can be computed in $O(|V_t|n)$ time.*

Theorem 5. *The TS problem is monotonic.*

4 Searching Trees with Sources and Targets

Now we study the STS problem. Recall that in this problem some searchers start on sources, and the search ends with some searchers occupying the targets. As in Section 2, we will actually study a slightly more general problem in order to make our analysis simpler:

Partial-Clear Source Target Searching (PSTS)
Instance: A tree T with a set of cleared edges $E_c \subseteq E(T)$, a set of source vertices $V_s \subseteq V(T)$, and a set of target vertices $V_t \subseteq V$, such that $V_t \cap V_s = \emptyset$, and each $v \in V_s$ is initially occupied by exactly one searcher.
Question: Is there a search strategy of T using k additional searchers such that once a searcher occupies a vertex $v \in V_t$, v remains occupied by at least one searcher for the remainder of the search, and each vertex from V_s remains clear during the entire search strategy?

We denote the search number associated with this searching problem as $psts(T, V_s, V_t, E_c)$. Our algorithm for the PSTS problem is a modified version of the algorithm presented in Section 2. The modifications take into account the addition of target vertices. Unlike the algorithm for the PSS problem, our algorithm for the PSTS problem does not necessarily first clear the subtree from $T \ominus V_s$ with the smallest search number. Instead, the algorithm first clears the subtree from $T \ominus V_s$ with the smallest search number *that does not contain a target vertex.* If all subtrees in $T \ominus V_s$ contain targets, then these subtrees are searched in increasing order by the number of target vertices they contain. Figure 3 illustrates why we must do this. In this example, $s(T_3) = 10$. If T_1 is cleared first then an additional searcher must be left behind to occupy v_1. To clear the remaining tree requires at least 10 additional searchers, resulting in a total of 11 additional searchers. The situation is the same if T_2 is cleared first. If T_3 is cleared first using 10 additional searchers, then T_1 can then be cleared using 2 of these additional searchers, leaving one behind on v_1, and then T_2 can be cleared using 2 more additional searchers, resulting in a total of 10 additional searchers.

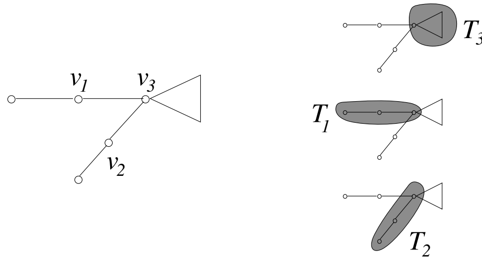

Fig. 3. A PSTS problem where $v_1, v_2 \in V_t$, $v_3 \in V_s$, and $s(T_3) = 10$

Now we present the new "reposition" algorithm that takes into account target vertices. A searcher λ occupying a vertex $v \in V_t$ is called *frozen* if it is the only searcher occupying v. The `Reposition-T` algorithm behaves the same as the `Reposition` algorithm from Section 2, except that frozen searchers are ignored (see Figure 4). We claim the following:

Lemma 4. *Let (T, V_s, V_t, E_c) be an instance of the PSTS problem. If V_s' and E_c' are results of running the* `Reposition-T` *algorithm, then $psts(T, V_s', V_t, E_c') \leq psts(T, V_s, V_t, E_c) + f$, where f is the number of starting searchers removed by the* `Reposition-T` *algorithm.*

Now we describe the main algorithm for computing $psts(T, V_s, V_t, E_c)$. The main difference between this algorithm and the one presented in Section 2 is that we must now take into account the target vertices. This is reflected in the choice that the algorithm makes with regard to the subtree that it chooses to clear first. Our algorithm is given in Figure 5.

Intuitively, the following Lemma shows that it is correct for the algorithm to clear T_1 first. In the following, T_{source} defined in step 4 of the `ST-ST` algorithm for the instance (T, V_s, V_t, E_c).

Algorithm Reposition-T(T, V_s, V_t, E_c) (Reposition with Targets)

1. While there exists a non-frozen moveable searcher λ on vertex $v \in V_s$:
   (a) Let vu be the only dirty edge incident with v.
   (b) Slide λ from v to u, clearing the edge vu, and $V_s \leftarrow (V_s - \{v\}) \cup \{u\}$, $E_c \leftarrow E_c \cup \{u\}$.
   (c) If u contains two searchers then remove λ and $f \leftarrow f + 1$.
2. While there exists a non-frozen searcher λ on a clear vertex $v \in V_s$:
   (a) Remove λ and $V_s \leftarrow V_s - \{v\}$.
   (b) $f \leftarrow f + 1$.

Fig. 4. The algorithm for repositioning the starting searchers when the tree contains targets

Algorithm ST-ST(T, V_s, V_t, E_c) (Search Tree with Sources and Targets)

1. If T is clear then return 0.
2. Call Reposition-T(T, V_s, V_t, E_c).
3. Compute $T \ominus V_s$, discarding all clear subtrees, to obtain $\{T_1, T_2, ..., T_k\}$.
4. Partition $T_1, T_2, \ldots, T_k$ into two sets T_{source} and T_{target} such that $T_i \in T_{target}$ if and only if T_i contains a target vertex. Sort T_{source} in increasing order by search number, and sort T_{target} in increasing order by the number of target vertices in the subtree. Furthermore, subtrees in T_{target} that have the same number of target vertices are sorted by their target search number.
5. If $T_{source} \neq \emptyset$ then,
   (a) Mark all edges in the first tree from T_{source}, say T_1, as clear. (explicitly clear this subtree if we are computing an actual search strategy, rather than just computing $psts(T, V_s, V_t, E_c)$.)
   (b) $s \leftarrow s(T_1)$ and go to step 7.
6. Otherwise, do the following:
   (a) Clear the first subtree from T_{target}, say T_1, using the algorithm from Section 3.
   (b) $s \leftarrow ts(T_1, V_t \cap V(T_1))$.
7. $s_{max} \leftarrow max\{s - f + |V_t \cap V(T_1)|,$ ST-ST$(T, V_s, V_t, E_c)\}$.
8. Return s_{max}.

Fig. 5. The algorithm for computing $psts(T, V_s, V_t, E_c)$

Lemma 5. *Let (T, V_s, V_t, E_c) be an instance of the PSTS problem. If $T_{source} \neq \emptyset$ then $psts(T, V_s, V_t, E_c) \geq s(T_1) - f$. Otherwise $psts(T, V_s, V_t, E_c) \geq ts(T_1) - f + |V_t \cap V(T_1)|$.*

As in Section 2, we can use induction and Lemma 5, along with Lemma 2 and 4 to show the following:

Theorem 6. *Let (T, V_s, V_t, E_c) be an instance of the PSTS problem with $|V(T)| = n$. The source target search number and the optimal search strategy of (T, V_s, V_t, E_c) can be computed in $O(|V_s||V_t|n)$ time.*

An analysis of the actions generated by the `Reposition-T` algorithm and Theorem 5 implies the following:

Theorem 7. *The PSTS problem is monotonic.*

5 Conclusion and Future Work

We have introduced and studied a new kind of graph searching problem involving sources and targets. We showed that each search model studied is monotonic. We have provided an $O(|V_s|n)$ time algorithm for the problem of searching a tree with $|V_s|$ sources, an $O(|V_t|n)$ time algorithm for searching a tree with $|V_t|$ targets, and an $O(|V_s||V_t|n)$ time algorithm for searching a tree with $|V_s|$ sources and $|V_t|$ targets. We conjecture that algorithms exist for all these problems that run in $O(n \log n)$ time. We are interested in studying source and target searching problems on larger classes of graphs. We are also interested in relating the source- and target-searching parameters to other graph parameters such as vertex separation and treewidth.

References

1. Dendris, N., Kirousis, L., Thilikos, D.: Fugitive-search games on graphs and related parameters. Theoretical Computer Science 172, 233–254 (1997)
2. Fomin, F., Petrov, N.: Pursuit-evasion and search problems on graphs. Congressus Numerantium 122, 47–58 (1996)
3. LaPaugh, A.: Recontamination does not help to search a graph. Journal of ACM 40, 224–245 (1993)
4. Megiddo, N., Hakimi, S., Garey, M., Johnson, D., Papadimitriou, C.: The complexity of searching a graph. Journal of ACM 35, 18–44 (1998)
5. Parsons, T.: Pursuit-evasion in a graph. In: Theory and Applications of Graphs. Lecture Notes in Mathematics, pp. 426–441. Springer, Heidelberg (1976)
6. Peng, S., Ho, C., Hsu, T., Ko, M., Tang, C.: Edge and Node Searching Problems on Trees. Theoretical Computer Science 240, 429–446 (2000)
7. Yang, B., Zhang, R., Cao, Y.: Searching Cycle-Disjoint Graphs. In: Dress, A.W.M., Xu, Y., Zhu, B. (eds.) COCOA 2007. LNCS, vol. 4616, pp. 32–43. Springer, Heidelberg (2007)