Using Tadpoles to Reduce Memory and Communication Requirements for Exhaustive, Breadth-First Search Using Distributed Computers

Gene Cooperman[1] and Michael Tselman[1]
gene@ccs.neu.edu and misha@ccs.neu.edu
College of Computer Science
Northeastern University
Boston, MA 02115 / U.S.A.

Abstract

A parallel variant of breadth-first search for distributed computing is presented. The variant allows exhaustive enumeration of elements of a search space (implicitly defined graph) in which the representation of all graph nodes would otherwise require more than the total available memory. This algorithm requires the use of a *tadpole* data structure to partition the search space into connected subgraphs, with each subgraph stored within the memory of a single processor. Thus, the graph of the nodes are stored in a way that adds greater spatial locality, thereby reducing communication among processors. The algorithm enumerates the tadpoles in a breadth-first manner while executing depth-first search within each tadpole. The search within each tadpole is reduced to tree search. A parameter is defined that allows a linear tradeoff in which memory and communication are each reduced linearly as CPU time grows. The result appears to fill a gap in the literature, which has concentrated on parallel depth-first search, but has been relatively sparse in the case of parallel breadth-first search.

1 Introduction

There are many applications in which one would want to exhaustively search for and save all elements of a very large search space, without duplication. Two generic cases are (1) enumeration of all elements of the search space that meet a certain criterion, and (2) saving a subset of the search space in a special structure for rapid retrieval or rapid query answering. The collected information is then fed into a domain-dependent application. Such applications often arise in such areas as operations research, cryptography, artificial intelligence, combinatorics, theorem proving, etc.

It should be emphasized here that we are assuming that the search space has no fast enumeration algorithm. If the search space had such an enumeration algorithm, then it would suffice to enumerate all elements and then feed only those elements satisfying the desired criterion into the domain-dependent application. Schroeder and Shamir formalize the term *polynomially enumerable*, and this has given rise to still another class of algorithms for exhaustive search [2, 15]. The result of this paper is particularly important for the case when the search space is not polynomially enumberable.

We define a *search space*, S, as follows. We assume that S is represented by a connected undirected graph. We assume that the graph S is given to us in an implicit representation, specified by the functions, $\ell(x)$ and $e_i(x)$ for $1 \le i \le \ell(x)$ and $x \in S$. For $x \in S$, let $\ell(x)$ be the *degree* of x (the number of neighbors of x) and let $e_i(x)$ be the i^{th} edge that contains x. The implicit representation assumes that one can compute edges $\{e_1(x), \ldots, e_{\ell(x)}(x)\}$ that connect the point x with each of its $\ell(x)$ neighbors. Further, given a point x and an edge $e_i(x)$, one can compute the other point, y, such that $(x, y) = e_i(x)$. Finally, one assumes that the implicit representation includes one or more initial points, where the exhaustive search begins.

In much of the existing literature, questions related to exhaustive search are either avoided or restricted to depth-first tree search for very large search spaces, where storage of the search space would require more than the available memory. A variant of breadth-first search on a distributed computer (e.g.: NOW or Network of Workstations) is described which overcomes this defect. The paper employs two key ideas in close combination. The first idea is to store only a certain subset of the search space, while maintaining rapid retrieval of all search space elements. This reduces the memory requirements below that for standard breadth-first search. The second idea is to make use of a partition of the search space into directed subgraphs called *tadpoles* [12], so that all elements in a given set in the partition are stored (represented) on a single processor. This locality allows us to reduce the communication costs below that of a more naive breadth-first algorithm.

The algorithm employs a user-specified parameter, m, allowing a linear tradeoff between CPU time and each of space and communication cost. Specifically, if ℓ is the average number of edges containing a node, $|S|$ is the total number of elements of the search space, and N is the number of processors, then we have the following asymptotic formulae:

	NAIVE	TADPOLE
AGGREG. CPU TIME:	$\Theta(\ell\|S\|)$	$\Theta(2\ell\|S\| + m\ell\|S\|)$
AGGREG. MEMORY:	$\Theta(\|S\|)$	$\Theta(\|S\|N/m)$
COMMUN. COST:	$\Theta(\ell\|S\|)$	$\Theta((\|S\|/m)\ln N)$

[1]Partially supported by NSF Grant CCR-9509783. This paper will appear in the Proceedings of SPAA-97.

In applications, m will typically be chosen much larger than N. The aggregate time and memory means the total across all processors. It is assumed that the number of tadpoles is larger than the number of processors, thus allowing all processors to be fully loaded.

For sufficiently large m, these formulae indicate the large reductions in memory and communication that are possible. For appropriate units of time, memory, and communication, the asymptotic coefficients in the $\Theta()$ can be taken as close to one. The asymptotic formulae hold under a certain heuristic assumption akin to the standard assumption of uniform hashing for hash functions. The assumption explained in section 4.1. Detailed derivations of these analyses can be found in section 7.

There appears to be little or no literature on parallel breadth-first, although strong activity in parallel depth-first search [1, 3, 9, 11, 13] can be observed. For these purposes, we define *breadth-first search* to be any search method that maintains a superset of the frontier of the currently known points. The lack of study of parallel breadth-first search in the literature seems to be because it typically requires a hash table storing representations of every element of the search space encountered. In contrast, depth-first search only requires storage proportional to the depth of search, and is otherwise independent of the size of the search space. Parallelism exacerbates this issue, because new nodes are encountered at a faster rate that would overload access to a distributed hash table. Yet, the sequential literature values breadth-first search in such variations as branch-and-bound (also known as A^* search) or bi-directional search [14, 16]. (In bi-directional search, one selects a source and destination in the search space, and attempts to find a shortest path between the two by using successively larger breadth-first style wavefronts extending both from the source and from the destination elements.) It should be noted that there is an extensive literature on parallel branch-and-bound (see, for example, [8]), but that this literature does not appear to address the issue of storing frontiers that are larger than available memory.

Note that our discussion is also distinct from the issues of self-organziing search, in which intelligent nodes on a network are simultaneously initialized and attempt to form a map of the network. See [10] for a breadth-first search approach to such self-organization.

The two ideas described here, storing only a subset of the search space, and the use of tadpole, are not new. It is the combination of the two that is novel. The methods described in this paper were motivated by the need of the authors to enumerate elements of large search spaces in applications of computational group theory [6, 7]. They were implemented using parallel tools (TOP-C [4] and STAR/MPI [5]), which allowed the parallelism of those applications to be easily implemented in a master-slave style. However, domain-specific information in those earlier applications removed the need for the tadpole. This paper uses the tadpole to provide a much more general approach to exhaustive search. The tadpole was invented by Richard Parker about 1986, although the name, "tadpole" was first used at a Nikolas conference about 1992. Unfortunately, we do not know of any prior, published description of the tadpole in the literature.

2 Naive Parallel Breadth-First Search

In this section we informally describe a natural adaptation of a breadth-first search algorithm to a distributed environment, against which the tadpole algorithm will be compared. The idea behind the adaptation is quite simple. Each point in a search space has an "owner" processor, which is computed by a hash-like function from the representation of the point to the set of processors according to a uniform distribution. Each processor maintains a local hash table (independent of the above hash-like function) to store all points owned by that processor. Thus, if a processor owns a point, it is easy for that processor to determine if it has previously encountered that point.

The search process is now very simple. Each processor begins with a small set of points "owned" by itself. These points are defined as the initial frontier of the breadth-first search. The frontier is then expanded by computing neighbors of the points on the current frontier. The point on the current frontier is then replaced by its neighbors. Those points that are "local" (i.e. owned by the processor that discovered it) are resolved locally. If the local point is known, it is discarded, and if the local point is new, it is added to the local hash array. If the point is not local, it is sent to to its "owner" processor and deleted from the frontier of the original processor. The owner processor is then able to resolve the point locally (by adding it to the frontier or deleting it from consideration. The algorithm stops when the frontiers on all processors become empty. This search strategy includes a natural load-balancing algorithm, since points of the search space tend to be approximately uniformly distributed among all processors.

Let $|S|$ be the total number of points, N the number of processors, and ℓ the average degree ($2|E|/|S|$, for $|E|$ the number of edges). An easy analysis shows that the aggregate computation time for the algorithm is $\Theta(\ell|S|)$, since for each point its ℓ neighbors are computed only once. (That happens on the processor that is the "owner" of the point). The total amount of communication is again $\ell|S|$ as every point is discovered on the average ℓ times (each time from a different neighbor), and it has to be sent out to the "owner" processor each of those ℓ times.

As we assume that $|S|$ is very large, the amount of communication that has to be performed for this algorithm can be beyond reasonable limits for slow networks. Further the semiconductor memory of the N processors may not suffice for large problems. It is these areas in which the tadpole approach provides better results, since it provides for flexible communication/computation/memory tradeoffs.

3 Overview of the Tadpole Algorithm

In this section, we provide a top-level overview of the algorithm. We first define the tadpole. The points of the search space can be partitioned into subsets, which we call *tadpoles*. The tadpole is a certain weakly connected, directed subgraph with outdegree one, corresponding to a subset of the partition.

In order to formally construct a tadpole structure we first convert the undirected search space graph S into a directed graph by replacing each edge with two directed edges — one edge in each direction. Every point x then has $\ell(x)$ outgoing edges which we call $\{e_1(x), \ldots, e_{\ell(x)}(x)\}$ and $\ell(x)$ incoming edges which we call $\{e_1^{-1}(x), \ldots, e_{\ell(x)}^{-1}(x)\}$.

Let $t\colon S \to [1\ldots]$, be a pseudo-random function from the set of points in S onto the integers such that $t(x) = i$, where $i \in [1, \ldots, \ell(x)]$. Now let us consider the directed subgraph consisting of all points of S and the edges $\{e_{t(x)}(x)\colon x \in S\}$. This graph is not necessarily even weakly connected. A *tadpole* is defined to be a weakly connected component of the directed *subgraph*. Figure 1 shows an example of a tadpole.

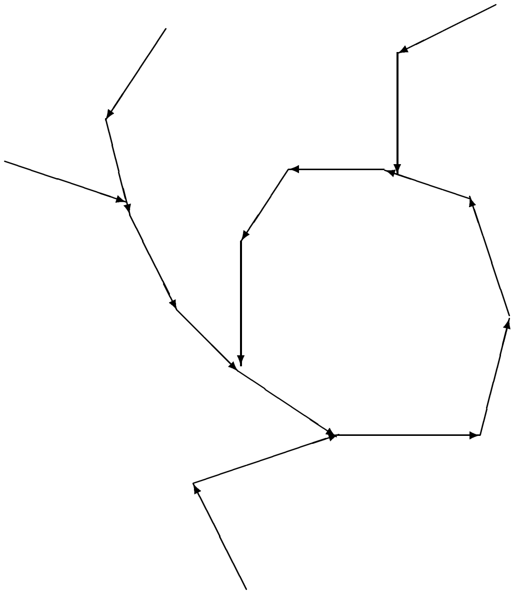

Figure 1: A tadpole

Our algorithm is designed for a distributed memory environment. We assume that the search space is large enough so that we cannot store all points on all processors. Yet, in order to prevent processors from duplicating each other's work and rediscovering the same tadpole, it is important for all processors to identify to each other which tadpoles they have found. Finally, we also want to keep communication costs low. So, for each tadpole, we define a subset of tadpole representatives and only these are communicated to other processors. Then each processor stores all tadpole representatives found so far on any processor. In order to understand the top-level portion of the algorithm it is sufficient to know that every tadpole has at least one representative and every path in the tadpole, if continued, eventually leads to a tadpole representative. We defer the formal definition of tadpole representatives until section 4.3.

3.1 Overview

3.1.1 Informal description

The algorithm proceeds in a breadth-first search manner over the set of tadpoles. The algorithm can be divided into two stages. In the first stage we select a number of points chosen at random on each processor and build tadpoles to which those points belong. (These points are arbitrary, and are not necessarily "owned" by the processor in the sense of section 2.) Whenever a tadpole is found, its representatives are broadcast to all other processors, thus insuring that no other processor explores the same tadpole in the future. If we choose sufficiently many initial points at random, there is a high probability that by the end of the first stage we discover most of the large tadpoles and therefore a majority of the points of the search space. After tadpoles for all initial points are found the algorithm enters the second stage.

In the second stage we need to find the remaining tadpoles. For simplicity, we assume that the remaining tadpoles contain sufficiently few remaining points, so that a single processor has the memory to store all of the remaining point. More sophisticated strategies are possible. The remaining points are found in two steps. In the first step we explore all neighbors of the tadpoles that have so far been found. I.e. For each point in the known tadpole we look at its neighbors and if they belong to yet unknown tadpoles we construct these tadpoles. While constructing these tadpoles we also look at their neighbors and if they are new we add them to the search frontier which is used in the second step. When all initial tadpoles are processed we are left with a yet unexplored frontier. The second step takes each point from the frontier and if it is still new creates its tadpole and adds its neighbors (if new) back to the frontier. This process continues until the frontier is empty. By that moment we can guarantee that for every point we explored all of its edges and, therefore, the exhaustive search is completed.

3.1.2 Pseudo-code

The pseudo-code assumes certain technical subroutines, such as traversal of a tadpole, that are described in section 5. Deferring those technical details allows us to concentrate on the key ideas.

In the first stage of the algorithm we start with a random set P of points from S, which can be selected by a random walk. Then the first stage proceeds as follows. The variable T_{reps} is the set of representatives of the given tadpole, and G is the set of tadpole representatives stored within the hash table of the current processor. Although several processes may happen to explore the same tadpole at the same time, this does no harm. When the number of tadpoles is large, the probability of such an event is small.

TadpoleStage-1

Let $G \leftarrow \emptyset$ [tadpole representatives]
Let $K \leftarrow \emptyset$ [one point from each tadpole]
For each $x \in P$
  If not `known`(x)
    $T_{\text{reps}} \leftarrow$ `Traverse-Tadpole`(x)
    $G \leftarrow G \cup T_{\text{reps}}$
    $K \leftarrow K \cup$ `first`(T_{reps})
    Broadcast T_{reps}
    [Receiving processes each execute $G \leftarrow G \cup T_{\text{reps}}$
      and $K \leftarrow K \cup$ `first`(T_{reps})]

The function `known()` is simple enough, so that we do not write separate pseudocode for it. The function `known()` checks if its argument is a known point (i.e. if it belongs to a known tadpole). In order to do that, the function follows a unique path in the tadpole until it reaches a tadpole representative. Then it checks if the representative point is already stored. This resolves the problem. The function `Traverse-Tadpole()` returns all representatives in a tadpole. It is described in section 5.

After the first stage of the algorithm is completed, we have a collection of tadpole representatives for most of the tadpoles (not necessarily all) stored in G. In order to reduce memory requirements of the algorithm we store only the tadpole representatives. Any tadpole can be reconstructed even from a single point, but it is useful to store multiple representatives so that the function `known()` can determine more efficiently if a tadpole has been seen before.

At the end of stage 1, "most" of the tadpoles have been found (assuming the initially chosen set of random points, P, was large enough). Representatives for each of those tadpoles are known. It remains to find the remaining tadpoles. The second stage of the algorithm proceeds as follows.

TadpoleStage-2

Let $F \leftarrow \emptyset$
For each $x \in K$
  $(T_{\text{reps}}, T_{\text{neighbors}}) \leftarrow$ `TraverseAndAddNeighbors(`x`)`
  $F \leftarrow F \cup T_{\text{neighbors}}$
While $F \neq \emptyset$
  extract x from F
  If not `known`(x)
    $(T_{\text{reps}}, T_{\text{neighbors}}) \leftarrow$ `TraverseAndAddNeighbors(`x`)`
    $G \leftarrow G \cup T_{\text{reps}}$
    $F \leftarrow F \cup T_{\text{neighbors}}$
    Broadcast T_{reps}
    [Receiving processes each execute $G \leftarrow G \cup T_{\text{reps}}$]

The function `TraverseAndAddNeighbors()` is a generalization of the function `Traverse-Tadpole()` and is also described in the section 5. It returns a pair of values, the first being a set of representatives for the traversed tadpole, and the second being a set of all points adjacent to the tadpole that are not yet known. We could not use the function `TraverseAndAddNeighbors()` in the first part of the algorithm because in the first part the largest tadpoles are most likely to be discovered and storing all their neighbors can be prohibitively expensive in terms of memory. However, by the time we start the second stage, most of the neighboring points will already belong to some known tadpoles and will be immediately discarded, thus keeping the dynamic memory requirements low.

It remains to formally define the tadpole representatives that were used in this section (section 3). We then provide pseudo-code for the various functions that traverse a tadpole.

4 Definitions

4.1 Tadpole

Recall the definition of tadpole in section 3. It is easy to see that every point in the subgraph has one outgoing edge and zero or more incoming edges. Since each point has a unique outgoing edge, this defines a dynamical system, $d_t : S \to S$, where $d_t(x) = y$ such that $e_{t(x)}(x) = (x, y)$. In our application of the tadpole, we will make the heuristic assumption:

Uniform tadpole assumption: The choice of the pseudo-random function t causes the dynamical system, d_t, to behave as if it were chosen at random among all dynamical systems on the node set, S.

If the function d_t is to be computable, at best it will only be pseudo-random. Further, this is close to requiring a random edge set on S, which is clearly not true. However, several of the pertinent statistics about tadpoles empirically appear to behave in implementations as if d_t were truly chosen at random. In this respect it is worth drawing an analogy with the uniform hashing assumption for hash functions. Although the assumption is not true for many of the hash functions used, those hash functions nevertheless have many of the same statistics as if the uniform hashing assumption were formally true.

As in any dynamical system on a finite set, the dynamical system partitions the finite set into subsets. The subsets are called *domains of attraction* in the context of dynamical systems. The points of each tadpole correspond to a domain of attraction. The name "tadpole" comes from the path followed by iterates of a typical point — i.e. the path begins with the tail of the tadpole and ends with a loop. (Since every point has an outgoing edge every path in the graph ends in a loop.) The set of points in the loop at the end is known as an *attractor cycle*. It is easy to see that each tadpole contains a unique attractor cycle.

4.2 Tadpole Root

Suppose that a point, x, belongs to the attractor cycle of one of the tadpoles. If one removes the outgoing edge from x, it is easy to see that this transforms the tadpole structure into a directed tree with root x. In agreement with the definition, each edge of the tree is directed from children toward their parents. For specificity, one chooses the *tadpole root*, x_{root}, to be the smallest point in the attractor cycle of the tadpole according to some total ordering.

The function `findroot()` given a point x walks down the tadpole edges until a cycle is reached and recognized. Then the smallest point on the cycle is selected.

`findroot`(x)
Let $i \leftarrow 0$, $y \leftarrow x$, and $z \leftarrow d_t(x)$
While $y \neq z$
  $y \leftarrow d_t(y)$
  $z \leftarrow d_t(d_t(z))$
$y_{min} \leftarrow y \leftarrow d_t(y)$
While $y \neq z$
  $y \leftarrow d_t(y)$
  [Compare y and y_{min} based on "name"]
  If $y < y_{min}$ **Set** $y_{min} \leftarrow y$
Return y_{min}

If d_t is costly, then one may prefer an alternative algorithm for `findroot()` based on using a hash table to record the entire trajectory of a single point, and then stopping when a previously seen point is encountered.

4.3 Tadpole Representatives

We select tadpole representatives as follows. We choose an integer m a priori such that it is large enough for $|S|/m$ points to fit into the memory of a single processor. We next define a pseudo-random uniform function from the set of points in S into an integer interval: $f_m(x) : S \to [0 \ldots m-1]$. The points $x \in S$ such that $f_m(x) = 0$ are defined to be *tadpole representatives.*

In addition, each tadpole has a unique *tadpole root*, defined in section 4.2. The tadpole roots are also defined to be *tadpole representatives.*

The function `representative()` walks down the tadpole edges until the first representative, y, is found that satisfies either $f_m(y) = 0$ or $y =$ `findroot`(y).

`representative`(x)
Let $i \leftarrow 0$, and $y_0 \leftarrow x$
While `(TRUE)`
  If $f_m(y_i) = 0$
    Return y_i
  If $y_i = y_j$ and $i > j \geq 0$
    Return $\min(y_j, y_{j+1}, \ldots, y_i)$
  $y_{i+1} \leftarrow d_t(y_i)$
  $i \leftarrow i + 1$

The number of tadpole representatives will be approximately $|S|/m + T$, where T is the total number of tadpoles.

The tadpole roots account for the T term in $|S|/m + T$. So even those tadpoles that do not have representatives defined by function $f_m()$ still have a representative. Representatives will play the following role in preventing the discovery of already known tadpoles: Whenever some point is examined as a potentially new point, one will follow a unique path in the tadpole to a tadpole representative. If this tadpole representative is already known, then the initial point in question is also known. (`known(representative(`x`))` iff `known(`x`)`.)

An alternative, simpler formulation of tadpole representatives would be to store every m^{th} point. We prefer the above formulation because it allows two processors to independently agree on whether a point is a tadpole representative. The processors can do this without reference to the hash table, which might be in some incomplete state. In some implementations, the hash table might even be distributed across all processors, in order to further save memory. The above formulation may also be cheaper in situations (a) where the computation can be kept in cache by avoiding a hash table, or (b) where hash lookup is expensive, perhaps because the hash table has overflowed RAM into virtual memory.

4.4 Tadpole Size

In our use of tadpoles, it will be important to guarantee that the average tadpole size is large compared to the parameter m defined in section 4.3. This is because, as discussed in section 4.3, the total number of tadpole representatives that need to be stored is $|S|/m + T$, where T is the number of tadpoles. If many tadpoles are of size less than m, then it will no longer be the case that only approximately $|S|/m$ points need be stored. In the appendix, one uses the uniform tadpole assumption to show that the average tadpole is of size at least $|S|/[(1/2)\ln|S| - \ln 2]$.

Although we do not specify the size of the user-defined parameter m, the efficiency of the algorithm depends on an assumption that the size of the typically encountered tadpole is greater than the size of m. The estimates in the appendix motivate why we expect this to be the case even for m as large as $|S|/\log|S|$.

The importance of the tadpole is precisely that it defines a connected subgraph of "large" size. Although the uniform tadpole assumption is not formally true, experience shows that it is close enough to true for the tadpoles still to be quite large. Depending on the search space, a significant minority of the tadpoles may be less than m. Section 6 sketches how to modify the function, $t()$, so as to reduce further the number of these smaller tadpoles. Section 8 provides brief comments on how the tadpole method can be adapted to the problem of finding a path between two arbitrary nodes in a search space.

5 Tadpole Traversal

As just observed, the tadpoles partition the search space into some number of graph components. The next algorithm allows one to traverse a tadpole by viewing the tadpole as a tree rooted at the unique point specified by `findroot()`.

We choose a depth-first approach in order to traverse the tree. This approach is well known. The only issue is that the edges of a tadpole structure are pointing in the wrong direction to perform a traversal beginning with the root point. Fortunately, since the original graph corresponding to our search space is undirected, the corresponding directed graph has the property that for each edge going in one direction we have an edge going in the opposite direction.

The problem is, given a point x in a tadpole, how do we find all other points y in the tadpole such that $d_t(y) = x$? We first compute all of the neighbors of x in the original graph. Then for each neighbor of x, z, we compute $d_t(z)$ (determined by the edge $e_{t(z)}(z)$) to see if it leads back to x. If so, (z, x) is an edge of the tadpole containing x. Those neighbors, z, for which $d_t(z) = x$ are the children of x in the tree.

The pseudo-code below formalizes the traversal of a tadpole. It requires two utility functions: `representative()` and `findroot()` described in sections 4.3 and 4.2.

`Traverse-Tadpole`(x)
$T_{\text{reps}} \leftarrow \emptyset$
$x_{\text{root}} \leftarrow \text{findroot}(x)$
`Traverse-Tadpole-Next`(x_{root})
Return T_{reps}

`Traverse-Tadpole-Next`(x)
If $x =$ `representative`(x)
    $T_{\text{reps}} \leftarrow \{x\} \cup T_{\text{reps}}$
For each $i \in [1, \ldots, l(x)] \backslash \{t(x)\}$
    Compute y such that $e_i(x) = (x, y)$
    If $d_t(y) = x$ and $y \neq x_{\text{root}}$
      `Traverse-Tadpole-Next`(y)
Return T_{reps}

The function `TraverseAndAddNeighbors()` below is a generalization of the function `Traverse-Tadpole()` in that in addition to constructing a tadpole it also computes all of its neighbors and collects those that are new in the set defined by the variable N. The function returns a pair consisting of the set of representatives for the tadpole and the set N that then main algorithm then adds to the global search frontier, F. Note the trick, that instead of adding the neighbor point itself to the set N we add the representative computed for this point. This provides for some savings later on, when points on the frontier are expanded. It also allows us to first do a test and save in F only those points that belong to unknown tadpoles.

`TraverseAndAddNeighbors`(x)
$T_{\text{reps}} \leftarrow \emptyset$
$T_{\text{neighbors}} \leftarrow \emptyset$
$x_{\text{root}} \leftarrow \text{findroot}(x)$
`TraverseAndAddNeighbors-Next`(x_{root})
Return $(T_{\text{reps}}, T_{\text{neighbors}})$

`TraverseAndAddNeighbors-Next`(x)
If $x =$ `representative`(x)
    $T_{\text{reps}} \leftarrow \{x\} \cup T_{\text{reps}}$
For each $i \in [1, \ldots, l(x)] \backslash \{t(x)\}$
    Compute y such that $e_i(x) = (x, y)$
    If $d_t(y) = x$ and $y \neq x_{\text{root}}$
      `TraverseAndAddNeighbors-Next`(y)
    Else
      $z \leftarrow$ `representative`(y)
      If not `known`(z)
        $T_{\text{neighbors}} \leftarrow T_{\text{neighbors}} \cup \{z\}$
Return $(T_{\text{reps}}, T_{\text{neighbors}})$

6 Avoiding Small Tadpoles

As seen earlier, if there are too many small tadpoles (tadpoles of size comparable to m or smaller), there will be more tadpoles in all, and so the algorithm will generate significantly more tadpole representatives. This then worsens the memory and communication requirements. For example, the dynamical system described in section 4.1 has the defect of giving rise to a certain fraction of tadpoles with two-point attractor cycles. Such tadpoles will tend to be small. This section describes how to patch the dynamical system, $d_t()$, so as to make it less likely to generate tadpoles with two-point attractor cycles. Similar techniques can be adapted to reduce the number of tadpoles with attractor cycles that are small but larger than two.

The key observation is that in a two-point attractor cycle, there will be two edges in the corresponding directed subgraph. Deleting one of the edges will not destroy the connectedness of the search space. Thus, if a dynamical system, $d_t()$, gives rise to a two-point attractor cycle, then one deletes the directed edge from the smaller point to the large point according to some total ordering on points. One then adds, for example, The edge e_i for $i = (t(x) + 1) \bmod t(l)$. This will eliminate most of the two-point attractor cycles. (It's possible that the new edge will create a new two-point attractor. But the number of two-point attractor cycles will have been greatly reduced.)

We do not give details of how tadpole traversal and the finding of tadpole neighbors would have to be modified. In brief, the idea is that if a two-point attractor cycle is found, then we examine which edge should be replaced, back up, and proceed using the new directed edge rather than the old edge.

Further, tadpole traversal requires us to follow directed edges in the backwards direction and then determine if that edge is part of a tadpole. In the current situation, the edge may not appear to be part of the tadpole according to the dynamical system, $d_t()$, but it may be an edge that was later added. In that case, a little extra work must be done to check for this case. So, if the current point is x, and we follow an edge backwards to encounter y, we must first check if $d_t(y) = x$. However, even if $d_t(y) \neq x$, we must see if $d_t(d_t(y)) = y$. In this latter case it may be that one of the edges from y was deleted and replaced by (y, x).

7 Analysis

In this section we present some of the estimates on the communication, computation and memory requirements of the algorithm. We also compare this algorithm with a naive breadth-first search method.

- **Communication:**
  It is easy to observe that the only communications performed by the algorithm are the broadcasts of tadpole representatives. Further, the majority of the representatives are broadcasted only once. The exceptions are the representatives of the tadpoles that happen to be discoverd on different processors at the same time which is an unlikely event according to section 3.1.2. In section 4.3 we already mentioned that the total number of tadpole representatives is approximately $|S|/m+T$, where T is the total number of tadpoles. The computation in the appendix ensures that the size of an average tadpole is reasonably large. This make the T term in the above summation relatively small compared with $|S|/m$. Different networks have different broadcast capabilities. Since in this paper we do not address specific implementation issues, we consider it safe to assume that the broadcast operation requires logarithmic time steps. There are implementations of broadcast with either higher or lower complexities, but this seems to be a common complexity for many implementations. As the total number of nodes broadcasted in the algorithm is $|S|/m + T$, the cost of the broadcast operation will be $\Theta((|S|/m + T) \log N)$. In the table used in the introduction we omit the term T as it will usually not have a major contribution to the total, and it is difficult to characterize in terms of input parameters.

- **Computation:**
  Choosing computation of a point as a unit of computation, we estimate the aggregate computation time for the algorithm. For every point in the search space the following actions occur during the algorithm: The point can be discovered at most ℓ times. (Once from each of its neighbors.) This is true in an amortized sense, since ℓ is the average degree, or average number of neighbors of each point. Each time a point is discovered we need to test if the point is already known. That is done by following the tadpole links down to the representative. By our choice of representatives, on the average a representative point will be found after m steps. Each step involves computing a point. So $m\ell$ steps are required for all neighbors of a point. The other action is when a point participates in the tadpole traversal. In that case again ℓ neighbors are considered. For points discovered on the second stage of the algorithm that will be all. However for the points found in the first stage the process will be duplicated, since traversal and adding neighbors are done in separate stages. Since we estimate that the majority of points will be found during the first stage, this will bring the upper bound to 2ℓ. Combining these values together and considering all points we get $\Theta(2\ell|S| + m\ell|S|)$.

- **Memory:**
  Memory requirements for our algorithm are very simple. We need to be able to replicate all tadpole representatives on all processors. That amounts to $\Theta((|S|/m+T)N)$. To compute the actual size one would multiply that number by the size of a single element in bytes. Again, the table in the introduction omits the term T.

For the naive breadth-first search algorithm we have already computed in section 2 that the aggregate communication time can be expressed as $\Theta(\ell|S|)$ and the total amount of communication is also $\Theta(\ell|S|)$. The aggregate memory requirement for the naive breadth first search is the memory needed to store the whole set S. Thus $\Theta(|S|)$ memory is required in all. The table in the introduction displays side by side resource requirements for both algorithms. It is easy to see that by varying the parameter m users have more flexibility in fitting a problem into the available resources when the tadpole algorithm is used. Larger problems can be solved and slow networks or limited memory do not create an insurpassable obstacle.

8 Another Search Problem

The ability to partition an implicit subspace into large subsets appears to have other applications to search. We sketch such an application. Consider the problem of finding paths between two arbitrary points of a search space. (This can also be used to test if two points are in the same connected component.) Again, we consider a large search space that cannot be stored in memory, and so a shortest path is likely to be impractical. However, the following algorithm may provide some path. The search space can be viewed as partitioned into many tadpoles. There is then an induced graph, in which we define a point to be a tadpole of the original graph. Two points are adjacent if and only if there is a path between the corresponding tadpoles in the original graph. In the induced graph, each point is likely to have very high degree, since a large tadpole will typically be adjacent to many others.

The two arbitrary points then correspond to two points in the induced graph. One then employs one's favorite search method within the induced graph to find a path. Within the induced graph, one has the problem of (1) recognizing old points and of (2) finding a random edge from a given point.

In order to recognize old points of the induced graph, it suffices to store tadpole representatives only on the attractor cycle. This is important, since we wish to avoid the expense of enumerating all points of a large tadpole. In order to find a random edge from a given point, it suffices to execute some random walk (or perhaps a walk along the attractor cycle of random length). One then chooses a random neighbor of that point. One also stores the last point seen within the original tadpole, x, and the neighboring point in the new tadpole, y. The points (x, y) are then associated with that edge. This last information will be important in "pulling back" from the induced graph.

Once one has a path in the induced graph, one "pulls back". To do this, we need to be able to "join" two adjacent edges e_1 and e_2 of the induced graph. If e_1 has associated with it the original points (x_1, y_1) and e_2 has associated with it the original points (x_2, y_2), then we need to find a path in a tadpole between y_1 and x_2. There will always be such a path, since e_1 and e_2 are adjacent. With this path in the original graph, one then has the possibility of using various heuristics to perturb it into a shorter path.

9 Conclusion

The tadpole appears to be a versatile data structure for search over large search spaces that do not fit inside memory. This note gave an application to exhaustive enumeration of a search space, and sketches one additional application to another search problem. The methodology appears attractive for still other applications to be considered for the future.

It should be remarked that an alternative formulation of the tadpole version is to save only one copy of the hash table, distributed across all processors. This allows further tradeoffs by lowering the value, m, while raising the size of the hash table. As with the standard version, each element of the hash table would be "owned" by a unique processor, determined by the hash table entry. The implications are clear, in lowering the CPU time (for smaller m) and lowering the total memory, while greatly raising the communication costs. Nevertheless, the communication costs would be raised still further, if not for a significant feature of our algorithm: The tadpole representative, `representative`(x) of a point, x, can be found without accessing the hash table. This alternative is expected to be competitive in situations where m would otherwise be very much larger than N, the number processors.

10 Acknowledgements

The authors wish to thank Klaus Lux and Steve Linton, each of whom described to us the tadpole method of Richard Parker and some of its history. The authors also wish to thank Boaz Patt-Shamir for valuable discussions, and to thank the referees for additional comments.

APPENDIX: Average Tadpole Size

Probability Analysis of Parker's Tadpole Technique
We shall show that the expected number of attractor cycles of a connected component $\mathcal{C}$ in a search space, S, is bounded above by

$$(1/2)\log|\mathcal{C}| - \log 2.$$

This in turn yields a corresponding lower bound on the average size of a connected component in S. In order to carry out a probability analysis, one makes use of the heuristic assumption that $x \mapsto x^{h(x)}$ acts on each connected component $\mathcal{C}$ as a random function with the image of x chosen from a uniform distribution. As a means to carrying out the probability analysis, we imagine an experiment in which we initially know nothing about the image $h(x)$, and we map out the action of $x \mapsto x^{h(x)}$ on successive points x. Let $X \subseteq \mathcal{C}$ be the set of points for which the image under h has already been seen. Initially, X is the empty set. Define $f(x)$ as $x^{h(x)}$. We imagine mapping out the images of the elements of $\mathcal{C}$ in an order dictated by the following procedure, Trial(). We repeatedly execute Trial(), until $X = \mathcal{C}$.

```
Procedure Trial(X)
LOOP:
        If X = C then Return(X)
        Let x ∈ C \ X (x arbitrary)
        Set U ← ∅
        While x ∉ X ∪ U
                Set U ← U ∪ {x}
                Set x ← f(x)
        [Either x ∈ U or x ∈ X]
        Set X ← X ∪ U
        If x ∈ U then Return(X)
Goto LOOP
```

We wish to show that the expected number of invocations of Trial() until $X = \mathcal{C}$ is bounded above by $(1/2)\log|\mathcal{C}| - \log 2$. Let $x_1 = x$ and let $x_i = f^{(i-1)}(x)$. Let $U_i = \{x_1, \ldots, x_i\}$. We shall make some probabilistic estimates in terms of P_i, the probability that Trial() terminates with $x_i \in U$ after only one iteration of LOOP, and in terms of Q_i, the probability that Trial() completes the first iteration of LOOP with $x_i \in X$ and so Trial() does not terminate (assuming $X \neq \mathcal{C}$).

If $x \in X \setminus U$ after one iteration of LOOP in Trial(), then the expected size of U at the end of the iteration is $\Pr(\exists i,\, x_i \in X,\, |U_{i-1}| = i-1,\, U_{i-1} \cap X = \emptyset) = \sum_{i=1}^{|\mathcal{C}|}(i-1)Q_i$, where

$$\begin{aligned} Q_i = &\ \Pr(x_i \in X \text{ and } |U_{i-1}| = i-1 \text{ and } U_{i-1} \cap X = \emptyset) \\ = &\ \Pr(x_i \in X \,\big|\, |U_{i-1}| = i-1 \text{ and } U_{i-1} \cap X = \emptyset) \cdot \\ &\ \Pr(|U_{i-1}| = i-1 \,\big|\, U_{i-1} \cap X = \emptyset) \cdot \\ &\ \Pr(U_{i-1} \cap X = \emptyset). \end{aligned}$$

On the other hand, if $x \in U \setminus X$ after one iteration of LOOP in Trial(), then Trial() terminates after that iteration. The probability that it terminates after one iteration is $\Pr(\exists i,\, x_i \in U_{i-1},\, |U_{i-1}| = i-1,\, U_i \cap X = \emptyset) = \sum_{i=1}^{|\mathcal{C}|} P_i$, where

$$\begin{aligned} P_i = & \Pr(x_i \in U_{i-1} \text{ and } |U_{i-1}| = i-1 \text{ and } U_{i-1} \cap X = \emptyset) \\ = & \Pr(x_i \in U_{i-1} \bigm| |U_{i-1}| = i-1 \text{ and } U_{i-1} \cap X = \emptyset) \cdot \\ & \Pr(|U_{i-1}| = i-1 \bigm| U_{i-1} \cap X = \emptyset) \cdot \\ & \Pr(U_{i-1} \cap X = \emptyset). \end{aligned}$$

Although we do not use the fact, note that $(\sum_{i=1}^{|\mathcal{C}|} P_i) + (\sum_{i=1}^{|\mathcal{C}|} Q_i) = 1$.

Since $\Pr(x_i \in X \bigm| |U_{i-1}| = i-1 \text{ and } U_{i-1} \cap X = \emptyset) = |X|/|\mathcal{C}|$ and $\Pr(x_i \in U_{i-1} \bigm| |U_{i-1}| = i-1 \text{ and } U_{i-1} \cap X = \emptyset) = (i-1)/|\mathcal{C}|$,

$$\frac{(i-1)Q_i}{P_i} = \frac{(i-1)|X|/|\mathcal{C}|}{(i-1)/|\mathcal{C}|} = |X|.$$

Let ΔX be the expected set difference for X before and after an invocation of Trial(). One can identify $1/\sum_{i=1}^{|\mathcal{C}|} P_i$ with the expected number of iterations of LOOP in Trial().

$$\begin{aligned} |\Delta X| \approx & \left(\frac{1}{\sum_{i=1}^{|\mathcal{C}|} P_i} - 1\right) \sum_{i=1}^{|\mathcal{C}|}(i-1)Q_i + \sum_{i=1}^{|\mathcal{C}|}(i-1)P_i \\ = & |X| - \sum_{i=1}^{|\mathcal{C}|}(i-1)Q_i + \sum_{i=1}^{|\mathcal{C}|}(i-1)P_i \le |X| + \sqrt{|\mathcal{C}|} \end{aligned}$$

where $|X|$ refers to the size before an invocation of Trial(). There is an easy argument that after the first invocation of Trial(), $|X| \approx \sqrt{|\mathcal{C}|}$. When $|\Delta X|/|\mathcal{C}|$ is small compared to 1, we can let $x = |X|$ and let t be the number of iterations. Under these conditions, the solution to the following differential equation provides an upper bound for x: $\frac{\mathrm{d}x}{\mathrm{d}t} \le (x + |\Delta X|) + \sqrt{|\mathcal{C}|} \le 2x + 2\sqrt{|\mathcal{C}|}$ with $x(0) = \sqrt{|\mathcal{C}|}$. (The right hand side, $(x + |\Delta X|) + \sqrt{|\mathcal{C}|}$, was chosen so as to bound $x + \sqrt{|\mathcal{C}|}$ over the interval $[x(t), x(t+1)]$.) So, the number of iterations (and the expected number of cycles in a connected component) when $x(t) = |\mathcal{C}|$ is bounded above by $\log(2|\mathcal{C}| + 2\sqrt{|\mathcal{C}|}) - \log(4\sqrt{|\mathcal{C}|}) \to (1/2)\log|\mathcal{C}| - \log 2$.

References

[1] A. Aggarwal, R. Anderson and M-Y. Kao, "Parallel Depth-First Search in General Directed Graphs", *Siam Journal on Computing* **19**(2), 1990, pp. 397–409.

[2] H. Amirazizi and M. Hellman, "Time-Memory-Processor Trade-Offs", *IEEE Transactions on Information Theory* **34**(3), 1988, pp. 505–512.

[3] S.K. Chang, "Parallel Balancing of Binary Search Trees, *IEEE Trans. on Computers* **23**, 1974, pp. 441–445.

[4] G. Cooperman, "TOP-C: A Task-Oriented Parallel C Interface", 5^{th} *International Symposium on High Performance Distributed Computing* (HPDC-5), 1996, IEEE Press, pp. 141–150.

[5] G. Cooperman, "STAR/MPI: Binding a Parallel Library to Interactive Symbolic Algebra Systems", *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '95)*, ACM Press, pp. 126–132.

[6] G. Cooperman, L. Finkelstein, M. Tselman and B. York, "Constructing Permutation Representations for Matrix Groups", *J. of Symbolic Computation*, to appear.

[7] G. Cooperman and M. Tselman, "New Sequential and Parallel Algorithms for Generating High Dimension Hecke Algebras using the Condensation Technique", *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '96)*, ACM Press, 1996, pp. 155–160.

[8] A. Ferreira and J. Rolim, eds., *Parallel Algorithms for Irregularly Structured Problems* (Proceedings of IRREGULAR '95), Lecture Notes in Computer Science **980**, 1995, pp. 363–409.

[9] V. Kumar and V. Rao, "Parallel depth-first search II: Analysis" *Int. J. of Parallel Programming* **16**(6), 1987, pp. 479–499.

[10] S.A.M. Makki, "Efficient distributed breadth-first search algorithm", Computer Communications **19**, 1996, pp. 628–636.

[11] S.A.M. Makki and G. Havas, "Distributed algorithms for depth-first search", *Information Processing Letters* **60**(1), Oct. 14, 1996, pp. 7–12.

[12] R. Parker, oral communication.

[13] V. Rao and V. Kumar, "Parallel depth-first search I: Implementation" *Int. J. of Parallel Programming* **16**(6), 1987, pp. 501–519.

[14] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995.

[15] R. Schroeppel and A. Shamir, "A $T = O(2^{n/2})$, $S = O(2^{n/4}$ Algorithm for Certain NP-Complete Problems", *SIAM J. Computing* **10**(3), 1981, pp. 456–464.

[16] Tanimoto, *The Elements of Artificial Intelligence Using Common Lisp*, Second Edition, Freeman, 1995.